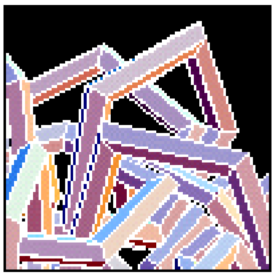
Contents

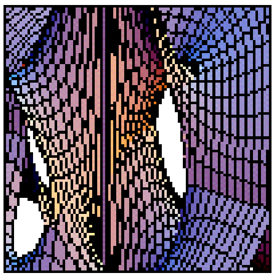
▹ ▹ **References**

P R O G R A M M I N G

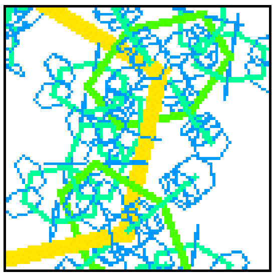


CHAPTER 1 Introduction to Mathematica

▹ ▹ **References**



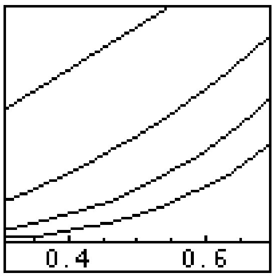
CHAPTER 2
Structure of Mathematica Expressions



CHAPTER **3**

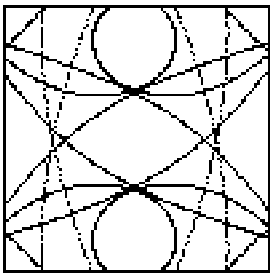
Definitions and Properties of Functions

▹ ▹ **References**



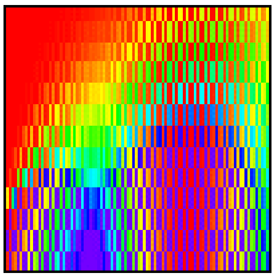
CHAPTER ***4***

Meta-Mathematica



CHAPTER **5**

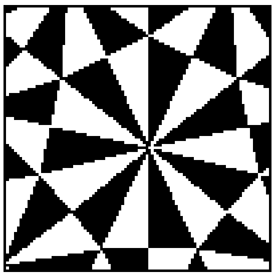
Restricted Patterns and Replacement Rules



CHAPTER **6**

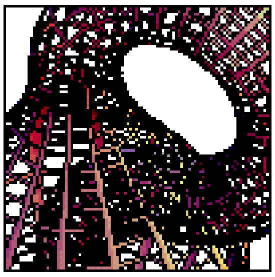
Operations on Lists, and Linear Algebra

G R A P H I C S

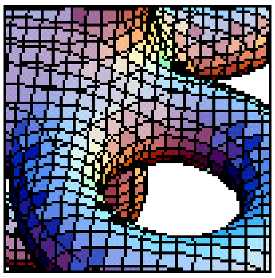


CHAPTER **1**

Two–Dimensional Graphics

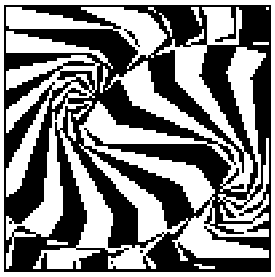


CHAPTER **2**
Three–Dimensional Graphics



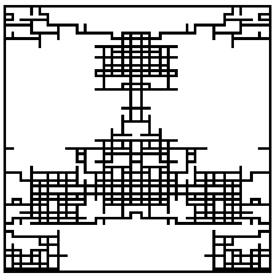
CHAPTER 3 *Contour and Density Plots*

N U M E R I C S



CHAPTER 1 Numerical Computations

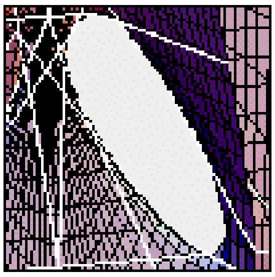
▹ ▹ References



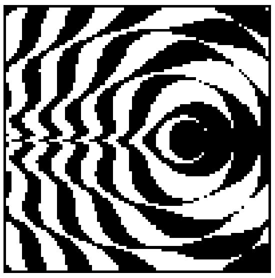
CHAPTER **2**

Computations with Exact Numbers

S Y M B O L I C S



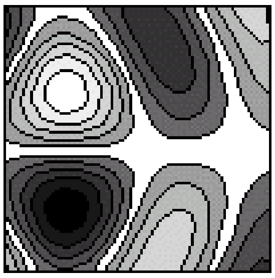
CHAPTER 1 Symbolic Computations



CHAPTER ***2***

Classical Orthogonal Polynomials

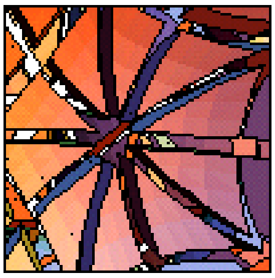
▹ ▹ References



CHAPTER 3 *Classical Special Functions*

▹ ▹ **References**

A P P E N D I C E S



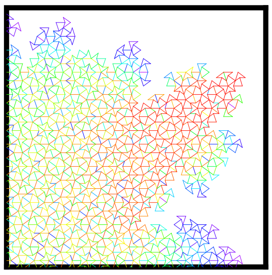
APPENDIX A

General References to Computer Algebra and to Mathematica

APPENDIX B (from http://www.mathematicaguidebooks.org)

The Front End, the Help Browser, Notebooks, Stylesheets, Cells, Typesetting, Buttons, Boxes, and All That

A D D I T I O N S



ADDITIONS FROM THE WEBSITE http://www.mathematicaguidebooks.org

Additional Exercises and Solutions

INTRODUCTION AND ORIENTATION

to *The Mathematica GuideBooks*

Language Concepts—Programming Examples—
Visualization Demos—Scientific Applications

0.1 Overview

■ 0.1.1 Content Summaries

The *Mathematica GuideBooks* are published as four independent books: *The Mathematica GuideBook to Programming, The Mathematica GuideBook to Graphics, The Mathematica GuideBook to Numerics*, and *The Mathematica Guide-Book to Symbolics.*

■ The Programming volume deals with the structure of *Mathematica* expressions and with *Mathematica* as a programming language. This volume includes the discussion of the hierarchical construction of all *Mathematica* objects out of symbolic expressions (all of the form *head*[*argument*]), the ultimate building blocks of expressions (numbers, symbols, and strings), the definition of functions, the application of rules, the recognition of patterns and their efficient application, the order of evaluation, program flows and program structure, the manipulation of lists (the universal container for *Mathematica* expressions of all kinds), as well as a number of topics specific to the *Mathematica* programming language. Various programming styles, especially *Mathematica*'s powerful functional programming constructs, are covered in detail.

■ The Graphics volume deals with *Mathematica*'s two-dimensional (2D) and three-dimensional (3D) graphics. The chapters of this volume give a detailed treatment on how to create images from graphics primitives, such as points, lines, and polygons. This volume also covers graphically displaying functions given either analytically or in discrete form. A number of images from the *Mathematica* Graphics Gallery are also reconstructed. Also discussed is the generation of pleasing scientific visualizations of functions, formulas, and algorithms. A variety of such examples are given.

■ The Numerics volume deals with *Mathematica*'s numerical mathematics capabilities—the indispensable sledgehammer tools for dealing with virtually any "real life" problem. The arithmetic types (fast machine, exact integer and rational, verified high-precision, and interval arithmetic) are carefully analyzed. Fundamental numerical operations, such as compilation of programs, numerical Fourier transforms, minimization, numerical solution of equations, and ordinary/partial differential equations are analyzed in detail and are applied to a large number of examples in the main text and in the solutions to the exercises.

■ The Symbolics volume deals with *Mathematica*'s symbolic mathematical capabilities—the real heart of *Mathematica* and the ingredient of the *Mathematica* software system that makes it so unique and powerful. Structural and mathematical operations on systems of polynomials are fundamental to many symbolic calculations and are covered in detail. The solution of equations and differential equations, as well as the classical calculus operations, are exhaustively treated. In addition, this volume discusses and employs the classical orthogonal polynomials and special functions of mathematical physics. To demonstrate the symbolic mathematics power, a variety of problems from mathematics and physics are discussed.

The four *GuideBooks* contain about 25,000 *Mathematica* inputs, representing more than 75,000 lines of commented *Mathematica* code. (For the reader already familiar with *Mathematica*, here is a more precise measure: The `Leaf`⋱ `Count` of all inputs would be about 900,000 when collected in a list.) The *GuideBooks* also have more than 4,000 graphics, 150 animations, 11,000 references, and 1,000 exercises. More than 10,000 hyperlinked index entries and hundreds of hyperlinks from the overview sections connect all parts in a convenient way. The evaluated notebooks of all four volumes have a cumulative file size of about 20 GB. Although these numbers may sound large, the *Mathematica GuideBooks* actually cover only a portion of *Mathematica*'s functionality and features and give only a glimpse into the possibilities *Mathematica* offers to generate graphics, solve problems, model systems, and discover new identities, relations, and algorithms. The *Mathematica* code is explained in detail throughout all chapters. More than 13,000 comments are scattered throughout all inputs and code fragments.

■ 0.1.2 Relation of the Four Volumes

The four volumes of the *GuideBooks* are basically independent, in the sense that readers familiar with *Mathematica* programming can read any of the other three volumes. But a solid working knowledge of the main topics discussed in *The Mathematica GuideBook to Programming*—symbolic expressions, pure functions, rules and replacements, and list manipulations—is required for the Graphics, Numerics, and Symbolics volumes. Compared to these three volumes, the Programming volume might appear to be a bit "dry". But similar to learning a foreign language, before being rewarded with the beauty of novels or a poem, one has to sweat and study. The whole suite of graphical capabilities and all of the mathematical knowledge in *Mathematica* are accessed and applied through lists, patterns, rules, and pure functions, the material discussed in the Programming volume.

Naturally, graphics are the center of attention of the *The Mathematica GuideBook to Graphics*. While in the Programming volume some plotting and graphics for visualization are used, graphics are not crucial for the Programming volume. The reader can safely skip the corresponding inputs to follow the main programming threads. The Numerics and Symbolics volumes, on the other hand, make heavy use of the graphics knowledge acquired in the Graphics volume. Hence, the prerequisites for the Numerics and Symbolics volumes are a good knowledge of *Mathematica's* programming language and of its graphics system.

The Programming volume contains only a few percent of all graphics, the Graphics volume contains about two-thirds, and the Numerics and Symbolics volume, about one-third of the overall 4,000+ graphics. The Programming and Graphics volumes use some mathematical commands, but they restrict the use to a relatively small number (especially `Expand`, `Factor`, `Integrate`, `Solve`). And the use of the function `N` for numericalization is unavoidable for virtually any "real life" application of *Mathematica*. The last functions allow us to treat some mathematically not uninteresting examples in the Programming and Graphics volumes. In addition to putting these functions to work for nontrivial problems, a detailed discussion of the mathematics functions of *Mathematica* takes place exclusively in the Numerics and Symbolics volumes.

The Programming and Graphics volumes contain a moderate amount of mathematics in the examples and exercises, and focus on programming and graphics issues. The Numerics and Symbolics volumes contain a substantially larger amount of mathematics.

Although printed as four books, the fourteen individual chapters (six in the Programming volume, three in the Graphics volume, two in the Numerics volume, and three in the Symbolics volume) of the *Mathematica GuideBooks* form one organic whole, and the author recommends a strictly sequential reading, starting from Chapter 1 of the Programming volume and ending with Chapter 3 of the Symbolics volume for gaining the maximum benefit. The electronic component of each book contains the text and inputs from all the four *GuideBooks*, together with a comprehensive hyperlinked index. The four volumes refer frequently to one another.

■ 0.1.3 Chapter Structure

A rough outline of the content of a chapter is the following:

■ The main body discusses the *Mathematica* functions belonging to the chapter subject, as well their options and attributes. Generically, the author has attempted to introduce the functions in a “natural order”. But surely, one cannot be axiomatic with respect to the order. (Such an order of the functions is not unique, and the author intentionally has “spread out” the introduction of various *Mathematica* functions across the four volumes.) With the introduction of a function, some small examples of how to use the functions and comparisons of this function with related ones are given. These examples typically (with the exception of some visualizations in the Programming volume) incorporate functions already discussed. The last section of a chapter often gives a larger example that makes heavy use of the functions discussed in the chapter.

■ A programmatically constructed overview of each chapter functions follows. The functions listed in this section are hyperlinked to their attributes and options, as well as to the corresponding reference guide entries of *The Mathematica Book*.

■ A set of exercises and potential solutions follow. Because learning *Mathematica* through examples is very efficient, the proposed solutions are quite detailed and form up to 50% of the material of a chapter.

■ References end the chapter.

Note that the first few chapters of the Programming volume deviate slightly from this structure. Chapter 1 of the Programming volume gives a general overview of the kind of problems dealt with in the four *GuideBooks.* The second, third, and fourth chapters of the Programming volume introduce the basics of programming in *Mathematica*. Starting with Chapters 5 of the Programming volume and throughout the Graphics, Numerics, and Symbolics volumes, the above-described structure applies.

In the 14 chapters of the *GuideBooks* the author has chosen a “we” style for the discussions of how to proceed in constructing programs and carrying out calculations to include the reader intimately.

■ 0.1.4 Code Presentation Style

The typical style of a unit of the main part of a chapter is: Define a new function, discuss its arguments, options, and attributes, and then give examples of its usage. The examples are virtually always *Mathematica* inputs and outputs. The majority of inputs is in `InputForm` are the notebooks. On occasion `StandardForm` is also used. Although `StandardForm` mimics classical mathematics notation and makes short inputs more readable, for "program-like" inputs, `InputForm` is typically more readable and easier and more natural to align. For the outputs, `StandardForm` is used by default and occasionally the author has resorted to `InputForm` or `FullForm` to expose digits of numbers and to `TraditionalForm` for some formulas. Outputs are mostly not programs, but nearly always "results" (often mathematical expressions, formulas, identities, or lists of numbers rather than program constructs). The world of *Mathematica* users is divided into three groups, and each of them has a nearly religious opinion on how to format *Mathematica* code [1✱], [2✱]. The author follows the `InputForm` cult(ure) and hopes that the *Mathematica* users who do everything in either `StandardForm` or `TraditionalForm` will bear with him. If the reader really wants to see all code in either `StandardForm` or `TraditionalForm`, this can easily be done with the Convert To item from the Cell menu. (Note that the relation between `InputForm` and `StandardForm` is not symmetric. The `InputForm` cells of this book have been line-broken and aligned by hand. Transforming them into `StandardForm` or `TraditionalForm` cells works well because one typically does not line-break manually and align *Mathematica* code in these cell types. But converting `StandardForm` or `TraditionalForm` cells into `InputForm` cells results in much less pleasing results.)

In the inputs, special typeset symbols for *Mathematica* functions are typically avoided because they are not monospaced. But the author does occasionally compromise and use Greek, script, Gothic, and doublestruck characters.

In a book about a programming language, two other issues come always up: indentation and placement of the code.

■ The code of the *GuideBooks* is largely consistently formatted and indented. There are no strict guidelines or even rules on how to format and indent *Mathematica* code. The author hopes the reader will find the book's formatting style readable. It is a compromise between readability (mental parsabililty) and space conservation, so that the printed version of the *Mathematica GuideBook* matches closely the electronic version.

■ Because of the large number of examples, a rather imposing amount of *Mathematica* code is presented. Should this code be present only on the disk, or also in the printed book? If it is in the printed book, should it be at the position where the code is used or at the end of the book in an appendix? Many authors of *Mathematica* articles and books have strong opinions on this subject. Because the main emphasis of the *Mathematica GuideBooks* is on *solving* problems *with Mathematica* and not on the actual problems, the *GuideBooks* give all of the code at the point where it is needed in the printed book, rather than "hiding" it in packages and appendices. In addition to being more straightforward to read and conveniently allowing us to refer to elements of the code pieces, this placement makes the correspondence between the printed book and the notebooks close to 1:1, and so working back and forth between the printed book and the notebooks is as straightforward as possible.

0.2 Requirements

■ 0.2.1 Hardware and Software

Throughout the *GuideBooks*, it is assumed that the reader has access to a computer running a current version of *Mathematica* (version 5.0/5.1 or newer). For readers without access to a licensed copy of *Mathematica*, it is possible to view all of the material on the disk using a trial version of *Mathematica*. (A trial version is downloadable from http://www.wolfram.com/products/mathematica/trial.cgi.)

The files of the *GuideBooks* are relatively large, altogether more than 20 GB. This is also the amount of hard disk space needed to store uncompressed versions of the notebooks. To view the notebooks comfortably, the reader's computer needs 128 MB RAM; to evaluate the evaluation units of the notebooks 1 GB RAM or more is recommended.

In the *GuideBooks*, a large number of animations are generated. Although they need more memory than single pictures, they are easy to create, to animate, and to store on typical year-2005 hardware, and they provide a lot of joy.

■ 0.2.2 Reader Prerequisites

Although prior *Mathematica* knowledge is not needed to read *The Mathematica GuideBook to Programming*, it is assumed that the reader is familiar with basic actions in the *Mathematica* front end, including entering Greek characters using the keyboard, copying and pasting cells, and so on. Freely available tutorials on these (and other) subjects can be found at http://library.wolfram.com.

For a complete understanding of most of the *GuideBooks* examples, it is desirable to have a background in mathematics, science, or engineering at about the bachelor's level or above. Familiarity with mechanics and electrodynamics is assumed. Some examples and exercises are more specialized, for instance, from quantum mechanics, finite element analysis, statistical mechanics, solid state physics, number theory, and other areas. But the *GuideBooks* avoid very advanced (but tempting) topics such as renormalization groups [6✱], parquet approximations [27✱], and modular moonshines [14✱]. (Although *Mathematica* can deal with such topics, they do not fit the character of the *Mathematica GuideBooks* but rather the one of a *Mathematica Topographical Atlas* [a monumental work to be carried out by the *Mathematica*–Bourbakians of the 21st century]).

Each scientific application discussed has a set of references. The references should easily give the reader both an overview of the subject and pointers to further references.

0.3 What the GuideBooks Are and What They Are Not

■ 0.3.1 Doing Computer Mathematics

As discussed in the Preface, the main goal of the *GuideBooks* is to demonstrate, showcase, teach, and exemplify scientific problem solving with *Mathematica*. An important step in achieving this goal is the discussion of *Mathematica* functions that allow readers to become fluent in programming when creating complicated graphics or solving scientific problems. This again means that the reader must become familiar with the most important programming, graphics, numerics, and symbolics functions, their arguments, options, attributes, and a few of their time and space complexities. And the reader must know which functions to use in each situation.

The *GuideBooks* treat only aspects of *Mathematica* that are ultimately related to “doing mathematics”. This means that the *GuideBooks* focus on the functionalities of the kernel rather than on those of the front end. The knowledge required to use the front end to work with the notebooks can easily be gained by reading the corresponding chapters of the online documentation of *Mathematica*. Some of the subjects that are treated either lightly or not at all in the *Guide-Books* include the basic use of *Mathematica* (starting the program, features, and special properties of the notebook front end [16✱]), typesetting, the preparation of packages, external file operations, the communication of *Mathematica* with other programs via *MathLink*, special formatting and string manipulations, computer- and operating system-specific operations, audio generation, and commands available in various packages. “Packages” includes both, those distributed with *Mathematica* as well as those available from the *Mathematica* Information Center (http://library.wolfram.com/infocenter) and commercial sources, such as MathTensor for doing general relativity calculations (http://smc.vnet.net/MathTensor.html) or FeynCalc for doing high-energy physics calculations (http://www.feyncalc.org). This means, in particular, that probability and statistical calculations are barely touched on because most of the relevant commands are contained in the packages. The *GuideBooks* make little or no mention of the machine-dependent possibilities offered by the various *Mathematica* implementations. For this information, see the *Mathematica* documentation.

Mathematical and physical remarks introduce certain subjects and formulas to make the associated *Mathematica* implementations easier to understand. These remarks are not meant to provide a deep understanding of the (sometimes complicated) physical model or underlying mathematics; some of these remarks intentionally oversimplify matters.

The reader should examine all *Mathematica* inputs and outputs carefully. Sometimes, the inputs and outputs illustrate little-known or seldom-used aspects of *Mathematica* commands. Moreover, for the efficient use of *Mathematica*, it is very important to understand the possibilities and limits of the built-in commands. Many commands in *Mathematica* allow different numbers of arguments. When a given command is called with fewer than the maximum number of arguments, an internal (or user-defined) default value is used for the missing arguments. For most of the commands, the maximum number of arguments and default values are discussed.

When solving problems, the *GuideBooks* generically use a “straightforward” approach. This means they are not using particularly clever tricks to solve problems, but rather direct, possibly computationally more expensive, approaches. (From time to time, the *GuideBooks* even make use of a “brute force” approach.) The motivation is that when solving new “real life” problems a reader encounters in daily work, the “right mathematical trick” is seldom at hand. Neverthe-less, the reader can more often than not rely on *Mathematica* being powerful enough to often succeed in using a straight-forward approach. But attention is paid to *Mathematica*-specific issues to find time- and memory-efficient implementa-tions—something that should be taken into account for any larger program.

As already mentioned, all larger pieces of code in this book have comments explaining the individual steps carried out in the calculations. Many smaller pieces of code have comments when needed to expedite the understanding of how they work. This enables the reader to easily change and adapt the code pieces. Sometimes, when the translation from traditional mathematics into *Mathematica* is trivial, or when the author wants to emphasize certain aspects of the code, we let the code “speak for itself”. While paying attention to efficiency, the *GuideBooks* only occasionally go into the computational complexity ([8✱], [40✱], and [7✱]) of the given implementations. The implementation of very large, complicated suites of algorithms is not the purpose of the *GuideBooks*. The *Mathematica* packages included with *Mathematica* and the ones at *MathSource* (http://library.wolfram.com/database/MathSource) offer a rich variety of self-study material on building large programs. Most general guidelines for writing code for scientific calculations (like descriptive variable names and modularity of code; see, e.g., [19✱] for a review) apply also to *Mathematica* programs.

The programs given in a chapter typically make use of *Mathematica* functions discussed in earlier chapters. Using commands from later chapters would sometimes allow for more efficient techniques. Also, these programs emphasize the use of commands from the current chapter. So, for example, instead of list operation, from a complexity point of view, hashing techniques or tailored data structures might be preferable. All subsections and sections are “self-con-tained” (meaning that no other code than the one presented is needed to evaluate the subsections and sections). The price for this “self-containedness” is that from time to time some code has to be repeated (such as manipulating polygons or forming random permutations of lists) instead of delegating such programming constructs to a package. Because this repetition could be construed as boring, the author typically uses a slightly different implementation to achieve the same goal.

■ 0.3.2 Programming Paradigms

In the *GuideBooks,* the author wants to show the reader that *Mathematica* supports various programming paradigms and also show that, depending on the problem under consideration and the goal (e.g., solution of a problem, test of an algorithm, development of a program), each style has its advantages and disadvantages. (For a general discussion concerning programming styles, see [3✱], [41✱], [23✱], [32✱], [15✱], and [9✱].) *Mathematica* supports a functional programming style. Thus, in addition to classical procedural programs (which are often less efficient and less elegant), programs using the functional style are also presented. In the first volume of the *Mathematica GuideBooks*, the programming style is usually dictated by the types of commands that have been discussed up to that point. A certain portion of the programs involve recursive, rule-based programming. The choice of programming style is, of course, partially (ultimately) a matter of personal preference. The *GuideBooks*’ main aim is to explain the operation, limits, and efficient application of the various *Mathematica* commands. For certain commands, this dictates a certain style of programming. However, the various programming styles, with their advantages and disadvantages, are not the main concern of the *GuideBooks*. In working with *Mathematica*, the reader is likely to use different programming styles depending if one wants a quick one-time calculation or a routine that will be used repeatedly. So, for a given implementation, the program structure may not always be the most elegant, fastest, or “prettiest”.

The *GuideBooks* are not a substitute for the study of *The Mathematica Book* [45✱] http://documents.wolfram.com/mathematica). It is impossible to acquire a deeper (full) understanding of *Mathematica* without a *thorough* study of this book (reading it twice from the first to the last page is highly recommended). It *defines* the language and the spirit of *Mathematica*. The reader will probably from time to time need to refer to parts of it, because not all commands are discussed in the *GuideBooks*. However, the story of what can be done with *Mathematica* does not end with the examples shown in *The Mathematica Book*. The *Mathematica GuideBooks* go beyond *The Mathematica Book*. They present larger programs for solving various problems and creating complicated graphics. In addition, the *GuideBooks* discuss a number of commands that are not or are only fleetingly mentioned in the manual (e.g., some specialized methods of mathematical functions and functions from the `Developer`` and `Experimental`` contexts), but which the author deems important. In the notebooks, the author gives special emphasis to discussions, remarks, and applications relating to several commands that are typical for *Mathematica* but not for most other programming languages, e.g., `Map`, `MapAt`, `MapIndexed`, `Distribute`, `Apply`, `Replace`, `ReplaceAll`, `Inner`, `Outer`, `Fold`, `Nest`, `NestList`, `FixedPoint`, `FixedPointList`, and `Function`. These commands allow to write exceptionally elegant, fast, and powerful programs. All of these commands are discussed in *The Mathematica Book* and others that deal with programming in *Mathematica* (e.g., [33✱], [34✱], and [42✱]). However, the author’s experience suggests that a deeper understanding of these commands and their optimal applications comes only after working with *Mathematica* in the solution of more complicated problems.

Both the printed book and the electronic component contain material that is meant to teach in detail how to use *Mathematica* to solve problems, rather than to present the underlying details of the various scientific examples. It cannot be overemphasized that to master the use of *Mathematica*, its programming paradigms and individual functions, the reader must experiment; this is especially important, insightful, easily verifiable, and satisfying with graphics, which involve manipulating expressions, making small changes, and finding different approaches. Because the results can easily be visually checked, generating and modifying graphics is an ideal method to learn programming in *Mathematica*.

0.4 Exercises and Solutions

■ 0.4.1 Exercises

Each chapter includes a set of exercises and a detailed solution proposal for each exercise. When possible, all of the purely *Mathematica*-programming related exercises (these are most of the exercises of the Programming volume) should be solved by every reader. The exercises coming from mathematics, physics, and engineering should be solved according to the reader's interest. The most important *Mathematica* functions needed to solve a given problem are generally those of the associated chapter.

For a rough orientation about the content of an exercise, the subject is included in its title. The relative degree of difficulty is indicated by level superscript of the exercise number (^{L1} indicates easy, ^{L2} indicates medium, and ^{L3} indicates difficult). The author's aim was to present understandable interesting examples that illustrate the *Mathematica* material discussed in the corresponding chapter. Some exercises were inspired by recent research problems; the references given allow the interested reader to dig deeper into the subject.

The exercises are intentionally not hyperlinked to the corresponding solution. The independent solving of the exercises is an important part of learning *Mathematica*.

■ 0.4.2 Solutions

The *GuideBooks* contain solutions to each of the more than 1,000 exercises. Many of the techniques used in the solutions are not just one-line calls to built-in functions. It might well be that with further enhancements, a future version of *Mathematica* might be able to solve the problem more directly. (But due to different forms of some results returned by *Mathematica*, some problems might also become more challenging.) The author encourages the reader to try to find shorter, more clever, faster (in terms of runtime as well complexity), more general, and more elegant solutions. *Doing* various calculations is the most effective way to learn *Mathematica*. A proper *Mathematica* implementation of a function that solves a given problem often contains many different elements. The function(s) should have sensibly named and sensibly behaving options; for various (machine numeric, high-precision numeric, symbolic) inputs different steps might be required; shielding against inappropriate input might be needed; different parameter values might require different solution strategies and algorithms, helpful error and warning messages should be available. The returned data structure should be intuitive and easy to reuse; to achieve a good computational complexity, nontrivial data structures might be needed, etc. Most of the solutions do not deal with all of these issues, but only with selected ones and thereby leave plenty of room for more detailed treatments; as far as limit, boundary, and degenerate cases are concerned, they represent an outline of how to tackle the problem. Although the solutions do their job in general, they often allow considerable refinement and extension by the reader.

The reader should consider the given solution to a given exercise as a proposal; quite different approaches are often possible and sometimes even more efficient. The routines presented in the solutions are not the most general possible, because to make them foolproof for every possible input (sensible and nonsensical, evaluated and unevaluated, numerical and symbolical), the books would have had to go considerably beyond the mathematical and physical framework of the *GuideBooks*. In addition, few warnings are implemented for improper or improperly used arguments. The graphics provided in the solutions are mostly subject to a long list of refinements. Although the solutions do work, they are often sketchy and can be considerably refined and extended by the reader. This also means that the provided solutions to the exercises programs are not always very suitable for solving larger classes of problems. To increase their applicability would require considerably more code. Thus, it is not guaranteed that given routines will work correctly on related problems. To guarantee this generality and scalability, one would have to protect the variables better, implement formulas for more general or specialized cases, write functions to accept different numbers of variables, add type-checking and error-checking functions, and include corresponding error messages and warnings.

To simplify working through the solutions, the various steps of the solution are commented and are not always packed in a `Module` or `Block`. In general, only functions that are used later are packed. For longer calculations, such as those in some of the exercises, this was not feasible and intended. The arguments of the functions are not always checked for their appropriateness as is desirable for robust code. But, this makes it easier for the user to test and modify the code.

0.5 The Books Versus the Electronic Components

■ 0.5.1 Working with the Notebooks

Each volume of the *GuideBooks* comes with a multiplatform DVD, containing fourteen main notebooks tailored for *Mathematica* 4 and compatible with *Mathematica* 5. Each notebook corresponds to a chapter from the printed books. (To avoid large file sizes of the notebooks, all animations are located in the Animations directory and not directly in the chapter notebooks.) The chapters (and so the corresponding notebooks) contain a detailed description and explanation of the *Mathematica* commands needed and used in applications of *Mathematica* to the sciences. Discussions on *Mathematica* functions are supplemented by a variety of mathematics, physics, and graphics examples. The notebooks also contain complete solutions to all exercises. Forming an electronic book, the notebooks also contain all text, as well as fully typeset formulas, and reader-editable and reader-changeable input. (Readers can copy, paste, and use the inputs in their notebooks.) In addition to the chapter notebooks, the DVD also includes a navigation palette and fully hyperlinked table of contents and index notebooks. The *Mathematica* notebooks corresponding to the printed book are fully evaluated. The evaluated chapter notebooks also come with hyperlinked overviews; these overviews are not in the printed book.

When reading the printed books, it might seem that some parts are longer than needed. The reader should keep in mind that the primary tool for working with the *Mathematica* kernel are *Mathematica* notebooks and that on a computer screen and there “length does not matter much”. The *GuideBooks* are basically a printout of the notebooks, which makes going back and forth between the printed books and the notebooks very easy. The *GuideBooks* give large examples to encourage the reader to investigate various *Mathematica* functions and to become familiar with *Mathematica* as a system for doing mathematics, as well as a programming language. Investigating *Mathematica* in the accompanying notebooks is the best way to learn its details.

To start viewing the notebooks, open the table of contents notebook TableOfContents.nb. *Mathematica* notebooks can contain hyperlinks, and all entries of the table of contents are hyperlinked. Navigating through one of the chapters is convenient when done using the navigator palette GuideBooksNavigator.nb.

When opening a notebook, the front end minimizes the amount of memory needed to display the notebook by loading it incrementally. Depending on the reader’s hardware, this might result in a slow scrolling speed. Clicking the “Load notebook cache” button of the GuideBooksNavigator palette speeds this up by loading the complete notebook into the front end.

For the vast majority of sections, subsections, and solutions of the exercises, the reader can just select such a structural unit and evaluate it (at once) on a year-2005 computer (≥512 MB RAM) typically in a matter of minutes. Some sections and solutions containing many graphics may need hours of computation time. Also, more than 50 pieces of code run hours, even days. The inputs that are very memory intensive or produce large outputs and graphics are in inactive cells which can be activated by clicking the adjacent button. Because of potentially overlapping variable names between various sections and subsections, the author advises the reader not to evaluate an entire chapter at once.

Each smallest self-contained structural unit (a subsection, a section without subsections, or an exercise) should be evaluated within one *Mathematica* session starting with a freshly started kernel. At the end of each unit is an input cell. After evaluating all input cells of a unit in consecutive order, the input of this cell generates a short summary about the entire *Mathematica* session. It lists the number of evaluated inputs, the kernel CPU time, the wall clock time, and the maximal memory used to evaluate the inputs (excluding the resources needed to evaluate the `Program` cells). These numbers serve as a guide for the reader about the to-be-expected running times and memory needs. These numbers can deviate from run to run. The wall clock time can be substantially larger than the CPU time due to other processes running on the same computer and due to time needed to render graphics. The data shown in the evaluated notebooks came from a 2.5 GHz Linux computer. The CPU times are generically proportional to the computer clock speed, but can deviate within a small factor from operating system to operating system. In rare, randomly occurring cases slower computers can achieve smaller CPU and wall clock times than faster computers, due to internal time-constrained simplification processes in various symbolic mathematics functions (such as `Integrate`, `Sum`, `DSolve`, …).

The Overview Section of the chapters is set up for a front end and kernel running on the same computer and having access to the same file system. When using a remote kernel, the directory specification for the package `Overview.m` must be changed accordingly.

References can be conveniently extracted from the main text by selecting the cell(s) that refer to them (or parts of a cell) and then clicking the “Extract References” button. A new notebook with the extracted references will then appear.

The notebooks contain color graphics. (To rerender the pictures with a greater color depth or at a larger size, choose Rerender Graphics from the Cell menu.) With some of the colors used, black-and-white printouts occasionally give low-contrast results. For better black-and-white printouts of these graphics, the author recommends setting the `Color`⋱ `Output` option of the relevant graphics function to `GrayLevel`. The notebooks with animations (in the printed book, animations are typically printed as an array of about 10 to 20 individual graphics) typically contain between 60 and 120 frames. Rerunning the corresponding code with a large number of frames will allow the reader to generate smoother and longer-running animations.

Because many cell styles used in the notebooks are unique to the *GuideBooks*, when copying expressions and cells from the *GuideBooks* notebooks to other notebooks, one should first attach the style sheet notebook GuideBooks-Stylesheet.nb to the destination notebook, or define the needed styles in the style sheet of the destination notebook.

■ 0.5.2 Reproducibility of the Results

The 14 chapter notebooks contained in the electronic version of the *GuideBooks* were run mostly with *Mathematica* 5.1 on a 2 GHz Intel Linux computer with 2 GB RAM. They need more than 100 hours of evaluation time. (This does not include the evaluation of the currently unevaluatable parts of code after the Make Input buttons.) For most subsections and sections, 512 MB RAM are recommended for a fast and smooth evaluation “at once” (meaning the reader can select the section or subsection, and evaluate all inputs without running out of memory or clearing variables) and the rendering of the generated graphic in the front end. Some subsections and sections need more memory when run. To reduce these memory requirements, the author recommends restarting the *Mathematica* kernel inside these subsections and sections, evaluating the necessary definitions, and then continuing. This will allow the reader to evaluate all inputs.

In general, regardless of the computer, with the same version of *Mathematica*, the reader should get the same results as shown in the notebooks. (The author has tested the code on Sun and Intel-based Linux computers, but this does not mean that some code might not run as displayed (because of different configurations, stack size settings, etc., but the disclaimer from the Preface applies everywhere). If an input does not work on a particular machine, please inform the author. Some deviations from the results given may appear because of the following:

■ Inputs involving the function `Random[…]` in some form. (Often `SeedRandom` to allow for some kind of reproducibility and randomness at the same time is employed.)

■ *Mathematica* commands operating on the file system of the computer, or make use of the type of computer (such inputs need to be edited using the appropriate directory specifications).

■ Calculations showing some of the differences of floating-point numbers and the machine-dependent representation of these on various computers.

■ Pictures using various fonts and sizes because of their availability (or lack thereof) and shape on different computers.

■ Calculations involving `Timing` because of different clock speeds, architectures, operating systems, and libraries.

■ Formats of results depending on the actual window width and default font size. (Often, the corresponding inputs will contain `Short`.)

Using anything other than *Mathematica* Version 5.1 might also result in different outputs. Examples of results that change form, but are all mathematically correct and equivalent, are the parameter variables used in underdetermined systems of linear equations, the form of the results of an integral, and the internal form of functions like `Interpolat⋱ ingFunction` and `CompiledFunction`. Some inputs might no longer evaluate the same way because functions from a package were used and these functions are potentially built-in functions in a later *Mathematica* version. *Mathematica* is a very large and complicated program that is constantly updated and improved. Some of these changes might be design changes, superseded functionality, or potentially regressions, and as a result, some of the inputs might not work at all or give unexpected results in future versions of *Mathematica*.

■ 0.5.3 Earlier Versions of the Notebooks

The first printing of the Programming volume and the Graphics volumes of the *Mathematica GuideBooks* were published in October 2004. The electronic components of these two books contained the corresponding evaluated chapter notebooks as well as unevaluated versions of preversions of the notebooks belonging to the Numerics and Symbolics volumes. Similarly, the electronic components of the Numerics and Symbolics volume contain the corresponding evaluated chapter notebooks and unevaluated copies of the notebooks of the Programming and Graphics volumes. This allows the reader to follow cross-references and look up relevant concepts discussed in the other volumes. The author has tried to keep the notebooks of the *GuideBooks* as up-to-date as possible. (Meaning with respect to the efficient and appropriate use of the latest version of *Mathematica*, with respect to maintaining a list of references that contains new publications, and examples, and with respect to incorporating corrections to known problems, errors, and mistakes). As a result, the notebooks of all four volumes that come with later printings of the Programming and Graphics volumes, as well with the Numerics and Symbolics volumes will be different and supersede the earlier notebooks originally distributed with the Programming and Graphics volumes. The notebooks that come with the Numerics and Symbolics volumes are genuine *Mathematica* Version 5.1 notebooks. Because most advances in *Mathematica* Version 5 and 5.1 compared with *Mathematica* Version 4 occurred in functions carrying out numerical and symbolical calculations, the notebooks associated with Numerics and Symbolics volumes contain a substantial amount of changes and additions compared with their originally distributed version.

0.6 Style and Design Elements

■ 0.6.1 Text and Code Formatting

The *GuideBooks* are divided into chapters. Each chapter consists of several sections, which frequently are further subdivided into subsections. General remarks about a chapter or a section are presented in the sections and subsections numbered 0. (These remarks usually discuss the structure of the following section and give teasers about the usefulness of the functions to be discussed.) Also, sometimes these sections serve to refresh the discussion of some functions already introduced earlier.

Following the style of *The Mathematica Book* [45✱], the *GuideBooks* use the following fonts: For the main text, Times; for *Mathematica* inputs and built-in *Mathematica* commands, Courier plain (like `Plot`); and for user-supplied arguments, Times italic (like $userArgument_1$). Built-in *Mathematica* functions are introduced in the following style:

`MathematicaFunctionToBeIntroduced[`*typeIndicatingUserSuppliedArgument(s)*`]`

is a description of the built-in command `MathematicaFunctionToBeIntroduced` upon its first appearance. A definition of the command, along with its parameters is given. Here, *typeIndicatingUserSuppliedArgument(s)* is one (or more) user-supplied expression(s) and may be written in an abbreviated form or in a different way for emphasis.

The actual *Mathematica* inputs and outputs appear in the following manner (as mentioned above, virtually all inputs are given in `InputForm`).

```
In[5]:= (* A comment. It will be/is ignored as Mathematica input:
          Return only one of the solutions *)
        Last[Solve[{x^2 - y == 1, x - y^2 == 1}, {x, y}]]
```

Out[6]= $$\left\{x \to -\frac{1}{3} + 4\left(\frac{2}{3\left(9+\sqrt{177}\right)}\right)^{2/3} + \frac{\left(9+\sqrt{177}\right)^{2/3}}{3\,2^{2/3}\,3^{1/3}},\; y \to -2\left(\frac{2}{3\left(9+\sqrt{177}\right)}\right)^{1/3} + \frac{\left(\frac{1}{2}\left(9+\sqrt{177}\right)\right)^{1/3}}{3^{2/3}}\right\}$$

When referring in text to variables of *Mathematica* inputs and outputs, the following convention is used: Fixed, nonpattern variables (including local variables) are printed in Courier plain (the equations solved above contained the variables `x` and `y`). User supplied arguments to built-in or defined functions with pattern variables are printed in Times italic. The next input defines a function generating a pair of polynomial equations in *x* and *y*.

```
In[7]:= equationPair[x_, y_] := {x^2 - y == 1, x - y^2 == 1}
```

x and *y* are pattern variables (usimng the same letters, but a different font from the actual code fragments `x_` and `y_`) that can stand for any argument. Here we call the function `equationPair` with the two arguments `u + v` and `w - z`.

```
In[8]:= equationPair[u + v, w - z]
```

Out[8]= $\{(u+v)^2 - w + z == 1,\; u + v - (w - z)^2 == 1\}$

Occasionally, explanation about a mathematics or physics topic is given before the corresponding *Mathematica* implementation is discussed. These sections are marked as follows:

Mathematical Remark: Special Topic in Mathematics or Physics

A *short* summary or review of mathematical or physical ideas necessary for the following example(s).

From time to time, *Mathematica* is used to analyze expressions, algorithms, etc. In some cases, results in the form of English sentences are produced programmatically. To differentiate such automatically generated text from the main text, in most instances such text is prefaced by "◦" (structurally the corresponding cells are of type `"PrintText"` versus `"Text"` for author-written cells).

Code pieces that either run for quite long, or need a lot of memory, or are tangent to the current discussion are displayed in the following manner.

```
mathematicaCodeWhichEitherRunsVeryLongOrThatIsVeryMemoryIntensive
OrThatProducesAVeryLargeGraphicOrThatIsASideTrackToTheSubjectUnder
Discussion
(* with some comments on how the code works *)
```

To run a code piece like this, click the Make Input button above it. This will generate the corresponding input cell that can be evaluated if the reader's computer has the necessary resources.

The reader is encouraged to add new inputs and annotations to the electronic notebooks. There are two styles for reader-added material: `"ReaderInput"` (a *Mathematica* input style and simultaneously the default style for a new cell) and `"ReaderAnnotation"` (a text-style cell type). They are primarily intended to be used in the `Reading` environment. These two styles are indented more than the default input and text cells, have a green left bar and a dingbat. To access the `"ReaderInput"` and `"ReaderAnnotation"` styles, press the system-dependent modifier key (such as Control or Command) and 9 and 7, respectively.

■ 0.6.2 References

Because the *GuideBooks* are concerned with the solution of mathematical and physical problems using *Mathematica* and are not mathematics or physics monographs, the author did not attempt to give complete references for each of the applications discussed [38✱], [20✱]. The references cited in the text pertain mainly to the applications under discussion. Most of the citations are from the more recent literature; references to older publications can be found in the cited ones. Frequently URLs for downloading relevant or interesting information are given. (The URL addresses worked at the time of printing and, hopefully, will be still active when the reader tries them.) References for *Mathematica*, for algorithms used in computer algebra, and for applications of computer algebra are collected in the Appendix A.

The references are listed at the end of each chapter in alphabetical order. In the notebooks, the references are hyperlinked to all their occurrences in the main text. Multiple references for a subject are not cited in numerical order, but rather in the order of their importance, relevance, and suggested reading order for the implementation given.

In a few cases (e.g., pure functions in Chapter 3, some matrix operations in Chapter 6), references to the mathematical background for some built-in commands are given—mainly for commands in which the mathematics required extends beyond the familiarity commonly exhibited by non-mathematicians. The *GuideBooks* do not discuss the algorithms underlying such complicated functions, but sometimes use *Mathematica* to "monitor" the algorithms.

References of the form *abbreviationOfAScientificField/yearMonthPreprintNumber* (such as quant-ph/0012147) refer to the arXiv preprint server [43✱], [22✱], [30✱] at http://arXiv.org. When a paper appeared as a preprint and (later) in a journal, typically only the more accessible preprint reference is given. For the convenience of the reader, at the end of these references, there is a Get Preprint button. Click the button to display a palette notebook with hyperlinks to the corresponding preprint at the main preprint server and its mirror sites. (Some of the older journal articles can be downloaded free of charge from some of the digital mathematics library servers, such as http://gdz.sub.uni-goettingen.de, http://www.emis.de, http://www.numdam.org, and http://dieper.aib.uni-linz.ac.at.)

As much as available, recent journal articles are hyperlinked through their digital object identifiers (http://www.doi.org).

■ 0.6.3 Variable Scoping, Input Numbering, and Warning Messages

Some of the *Mathematica* inputs intentionally cause error messages, infinite loops, and so on, to illustrate the operation of a *Mathematica* command. These messages also arise in the user's practical use of *Mathematica*. So, instead of presenting polished and perfected code, the author prefers to illustrate the potential problems and limitations associated with the use of *Mathematica* applied to "real life" problems. The one exception are the spelling warning messages `General::spell` and `General::spell1` that would appear relatively frequently because "similar" names are used eventually. For easier and less defocused reading, these messages are turned off in the initialization cells. (When working with the notebooks, this means that the pop-up window asking the user "Do you want to automatically evaluate all the initialization cells in the notebook?" should be evaluated should always be answered with a "yes".) For the vast majority of graphics presented, the picture is the focus, not the returned *Mathematica* expression representing the picture. That is why the `Graphics` and `Graphics3D` output is suppressed in most situations.

To improve the code's readability, no attempt has been made to protect all variables that are used in the various examples. This protection could be done with `Clear`, `Remove`, `Block`, `Module`, `With`, and others. Not protecting the variables allows the reader to modify, in a somewhat easier manner, the values and definitions of variables, and to see the effects of these changes. On the other hand, there may be some interference between variable names and values used in the notebooks and those that might be introduced when experimenting with the code. When readers examine some of the code on a computer, reevaluate sections, and sometimes perform subsidiary calculations, they may introduce variables that might interfere with ones from the *GuideBooks*. To partially avoid this problem, and for the reader's convenience, sometimes `Clear[`*sequenceOfVariables*`]`and `Remove[`*sequenceOfVariables*`]` are sprinkled throughout the notebooks. This makes experimenting with these functions easier.

The numbering of the *Mathematica* inputs and outputs typically does not contain all consecutive integers. Some pieces of *Mathematica* code consist of multiple inputs per cell; so, therefore, the line numbering is incremented by more than just 1. As mentioned, *Mathematica* should be restarted at every section, or subsection or solution of an exercise, to make sure that no variables with values get reused. The author also explicitly asks the reader to restart *Mathematica* at some special positions inside sections. This removes previously introduced variables, eliminates all existing contexts, and returns *Mathematica* to the typical initial configuration to ensure reproduction of the results and to avoid using too much memory inside one session.

■ 0.6.4 Graphics

In *Mathematica* 5.1, displayed graphics are side effects, not outputs. The actual output of an input producing a graphic is a single cell with the text `-Graphics-` or `-Graphics3D-` or `-GraphicsArray-` and so on. To save paper, these output cells have been deleted in the printed version of the *GuideBooks*.

Most graphics use an appropriate number of plot points and polygons to show the relevant features and details. Changing the number of plot points and polygons to a higher value to obtain higher resolution graphics can be done by changing the corresponding inputs.

The graphics of the printed book and the graphics in the notebooks are largely identical. Some printed book graphics use a different color scheme and different point sizes and line and edge thicknesses to enhance contrast and visibility. In addition, the font size has been reduced for the printed book in tick and axes labels.

The graphics shown in the notebooks are PostScript graphics. This means they can be resized and rerendered without loss of quality. To reduce file sizes, the reader can convert them to bitmap graphics using the Cell⟶Convert To⟶ Bitmap menu. The resulting bitmap graphics can no longer be resized or rerendered in the original resolution.

To reduce file sizes of the main content notebooks, the animations of the *GuideBooks* are not part of the chapter notebooks. They are contained in a separate directory.

0.6.5 Notations and Symbols

The symbols used in typeset mathematical formulas are not uniform and unique throughout the *GuideBooks*. Various mathematical and physical quantities (such as normals, rotation matrices, and field strengths) are used repeatedly in this book. Frequently the same notation is used for them, but depending on the context, also different ones are used, e.g. sometimes bold is used for a vector (such as **r**) and sometimes an arrow (such as $\vec{r}$). Matrices appear in bold or as doublestruck letters. Depending on the context and emphasis placed, different notations are used in display equations and in the *Mathematica* input form. For instance, for a time-dependent scalar quantity of one variable $\psi(t; x)$, we might use one of many patterns, such as `ψ[t][x]` (for emphasizing a parametric t-dependence) or `ψ[t, x]` (to treat t and x on an equal footing) or `ψ[t, {x}]` (to emphasize the one-dimensionality of the space variable x).

Mathematical formulas use standard notation. To avoid confusion with *Mathematica* notations, the use of square brackets is minimized throughout. Following the conventions of mathematics notation, square brackets are used for three cases: a) Functionals, such as $\mathcal{F}_t[f(t)](\omega)$ for the Fourier transform of a function $f(t)$. b) Power series coefficients, $\left[x^k\right](f(x))$ denotes the coefficient of x^k of the power series expansion of $f(x)$ around $x = 0$. c) Closed intervals, like $[a, b]$ (open intervals are denoted by (a, b)). Grouping is exclusively done using parentheses. Upper-case double-struck letters denote domains of numbers, $\mathbb{Z}$ for integers, $\mathbb{N}$ for nonnegative integers, $\mathbb{Q}$ for rational numbers, $\mathbb{R}$ for reals, and $\mathbb{C}$ for complex numbers. Points in $\mathbb{R}^n$ (or $\mathbb{C}^n$) with explicitly given coordinates are indicated using curly braces $\{c_1, \ldots, c_n\}$. The symbols $\wedge$ and $\vee$ for And and Or are used in logical formulas.

For variable names in formula- and identity-like *Mathematica* code, the symbol (or small variations of it) traditionally used in mathematics or physics is used. In program-like *Mathematica* code, the author uses very descriptive, sometimes abbreviated, but sometimes also slightly longish, variable names, such as `buildBrillouinZone` and `Fibonacci⋱ ChainMap`.

■ 0.6.6 Units

In the examples involving concepts drawn from physics, the author tried to enhance the readability of the code (and execution speed) by not choosing systems of units involving numerical or unit-dependent quantities. (For more on the choice and treatment of units, see [39✱], [4✱], [5✱], [10✱], [13✱], [11✱], [12✱], [36✱], [35✱], [31✱], [37✱], [44✱], [21✱], [25✱], [18✱], [26✱], [24✱].) Although *Mathematica* can carry units along with the symbols representing the physical quantities in a calculation, this requires more programming and frequently diverts from the essence of the problem. Choosing a system of units that allows the equations to be written without (unneeded in computations) units often gives considerable insight into the importance of the various parts of the equations because the magnitudes of the explicitly appearing coefficients are more easily compared.

■ 0.6.7 Cover Graphics

The cover graphics of the *GuideBooks* stem from the *Mathematica GuideBooks* themselves. The construction ideas and their implementation are discussed in detail in the corresponding *GuideBook*.

■ The cover graphic of the Programming volume shows 42 tori, 12 of which are in the dodecahedron’s face planes and 30 which are in the planes perpendicular to the dodecahedron’s edges. Subsections 1.2.4 of Chapter 1 discusses the implementation.

■ The cover graphic of the Graphics volume first subdivides the faces of a dodecahedron into small triangles and then rotates randomly selected triangles around the dodecahedron’s edges. The proposed solution of Exercise 1b of Chapter 2 discusses the implementation.

■ The cover graphic of the Numerics volume visualizes the electric field lines of a symmetric arrangement of positive and negative charges. Subsection 1.11.1 discusses the implementation.

■ The cover graphic of the Symbolics volume visualizes the derivative of the Weierstrass $\wp'$ function over the Riemann sphere. The “threefold blossoms” arise from the poles at the centers of the periodic array of period parallelograms. Exercise 3j of Chapter 2 discusses the implementation.

■ The four spine graphics show the inverse elliptic nome function q^{-1}, a function defined in the unit disk with a boundary of analyticity mapped to a triangle, a square, a pentagon, and a hexagon. Exercise 16 of Chapter 2 of the Graphics volume discusses the implementation.

0.7 Production History

The original set of notebooks was developed in the 1991–1992 academic year on an Apple Macintosh IIfx with 20 MB RAM using *Mathematica* Version 2.1. Over the years, the notebooks were updated to *Mathematica* Version 2.2, then to Version 3, and finally for Version 4 for the first printed edition of the Programming and Graphics volume of the *Mathematica GuideBooks* (published autumn 2004). For the Numerics and Symbolics volume, the *GuideBooks* notebooks were updated to *Mathematica* Version 5 in the second half of 2004. The electronic component. Historically, the first step in creating the book was the translation of a set of Macintosh notebooks used for lecturing and written in German into English by Larry Shumaker. This was done primarily by a translation program and afterward by manually polishing the English version. Then the notebooks were transformed into T_EX files using the program `nb2tex` on a NeXT computer. The resulting files were manually edited, equations prepared in the original German notebooks were formatted with T_EX, and macros were added corresponding to the design of the book. (The translation to T_EX was necessary because *Mathematica* Version 2.2 did not allow for book-quality printouts.) They were updated and refined for nearly three years, and then *Mathematica* 3 notebooks were generated from the T_EX files using a preliminary version of the program `tex2nb`. Historically and technically, this was an important step because it transformed all of the material of the *GuideBooks* into *Mathematica* expressions and allowed for automated changes and updates in the various editing stages. (Using the *Mathematica* kernel allowed one to process and modify the notebook files of these books in a uniform and time-efficient manner.) Then, the notebooks were expanded in size and scope and updated to *Mathematica* 4. In the second half of the year 2003, and first half of the year 2004, the *Mathematica* programs of the notebooks were revised to be compatible with *Mathematica* 5. In October 2004, the Programming and the Graphics volumes were published. In the last quarter of 2004, all four volumes of the *GuideBooks* were updated to be tailored for *Mathematica* 5.1 A special set of styles was created to generate the actual PostScript as printouts from the notebooks. All inputs were evaluated with this style sheet, and the generated PostScript was directly used for the book production. Using a little *Mathematica* program, the index was generated from the notebooks (which are *Mathematica* expressions), containing all index entries as cell tags.

0.8 Four General Suggestions

A reader new to *Mathematica* should take into account these four suggestions.

■ There is usually more than one way to solve a given problem using *Mathematica*. If one approach does not work or returns the wrong answer or gives an error message, make every effort to understand what is happening. Even if the reader has succeeded with an alternative approach, it is important to try to understand why other attempts failed.

■ Mathematical formulas, algorithms, and so on, should be implemented as directly as possible, even if the resulting construction is somewhat “unusual” compared to that in other programming languages. In particular, the reader should not simply translate C, Pascal, Fortran, or other programs line-by-line into *Mathematica*, although this is indeed possible. Instead, the reader should instead reformulate the problem in a clear mathematical way. For example, `Do`, `While`, and `For` loops are frequently unnecessary, convergence (for instance, of sums) can be checked by *Mathematica*, and `If` tests can often be replaced by a corresponding pattern. The reader should start with an exact mathematical description of the problem [28✸], [29✸]. For example, it does not suffice to know which transformation formulas have to be used on certain functions; one also needs to know how to apply them. “The power of mathematics is in its precision. The precision of mathematics must be used precisely.” [17✸]

■ If the exercises, examples, and calculation of the *GuideBooks* or the listing of calculation proposals from Exercise 1 of Chapter 1 of the Programming volume are not challenging enough or do not cover the reader's interests, consider the following idea, which provides a source for all kinds of interesting and difficult problems: The reader should select a built-in command and try to reconstruct it using other built-in commands and make it behave as close to the original as possible in its operation, speed, and domain of applicability, or even to surpass it. (Replicating the following functions is a serious challenge: `N`, `Factor`, `FactorInteger`, `Integrate`, `NIntegrate`, `Solve`, `DSolve`, `NDSolve`, `Series`, `Sum`, `Limit`, `Root`, `Prime`, or `PrimeQ`.)

■ If the reader tries to solve a smaller or larger problem in *Mathematica* and does not succeed, keep this problem on a "to do" list and periodically review this list and try again. Whenever the reader has a clear strategy to solve a problem, this strategy can be implemented in *Mathematica*. The implementation of the algorithm might require some programming skills, and by reading through this book, the reader will become able to code more sophisticated procedures and more efficient implementations. After the reader has acquired a certain amount of *Mathematica* programming familiarity, implementing virtually all "procedures" which the reader can (algorithmically) carry out with paper and pencil will become straightforward.

References

✱1 P. Abbott. *The Mathematica Journal* 4, 415 (2000).

✱2 P. Abbott. *The Mathematica Journal* 9, 31 (2003).

✱3 H. Abelson, G. Sussman. *Structure and Interpretation of Computer Programs*, MIT Press, Cambridge, MA, 1985. *BookLink (5)*

✱4 G. I. Barenblatt. *Similarity, Self-Similarity, and Intermediate Asymptotics*, Consultants Bureau, New York, 1979. *BookLink (3)*

✱5 F. A. Bender. *An Introduction to Mathematical Modeling*, Wiley, New York, 1978. *BookLink (3)*

✱6 G. Benfatto, G. Gallavotti. *Renormalization Group*, Princeton University Press, Princeton, 1995. *BookLink (2)*

✱7 L. Blum, F. Cucker, M. Shub, S. Smale. *Complexity and Real Computation*, Springer, New York, 1998. *BookLink*

✱8 P. Bürgisser, M. Clausen, M. A. Shokrollahi. *Algebraic Complexity Theory*, Springer, Berlin, 1997. *BookLink*

✱9 L. Cardelli, P. Wegner. *Comput. Surveys* 17, 471 (1985).

✱10 J. F. Carinena, M. Santander in P. W. Hawkes (ed.). *Advances in Electronics and Electron Physics* 72, Academic Press, New York, 1988.

✱11 E. A. Desloge. *Am. J. Phys.* 52, 312 (1984). *DOI-Link*

✱12 C. L. Dym, E. S. Ivey. *Principles of Mathematical Modelling*, Academic Press, New York, 1980. *BookLink (2)*

✱13 A. C. Fowler. *Mathematical Models in the Applied Sciences*, Cambridge University Press, Cambridge, 1997. *BookLink (2)*

✱14 T. Gannon. *arXiv:math.QA*/9906167 (1999). *Get Preprint*

✱15 R. J. Gaylord, S. N. Kamin, P. R. Wellin. *An Introduction to Programming with Mathematica*, TELOS/Springer-Verlag, Santa Clara, 1993. *BookLink (4)*

✱16 J. Glynn, T. Gray. *The Beginner's Guide to Mathematica Version 3*, Cambridge University Press, Cambridge, 1997. *BookLink*

✱17 D. Greenspan in R. E. Mickens (ed.). *Mathematics and Science*, World Scientific, Singapore, 1990. *BookLink*

✱18 G. W. Hart. *Multidimensional Analysis*, Springer-Verlag, New York, 1995. *BookLink*

✱19 A. K. Hartman, H. Rieger. *arXiv:cond-mat*/0111531 (2001). *Get Preprint*

✱20 M. Hazewinkel. *arXiv:cs.IR*/0410055 (2004). *Get Preprint*

✱21 E. Isaacson, M. Isaacson. *Dimensional Methods in Engineering and Physics*, Edward Arnold, London, 1975. *BookLink*

✱22 A. Jackson. *Notices Am. Math. Soc.* 49, 23 (2002).

✱23 R. D. Jenks, B. M. Trager in J. von zur Gathen, M. Giesbrecht (eds.). *Symbolic and Algebraic Computation*, ACM Press, New York, 1994. *DOI-Link*

✱24 C. G. Jesudason. *arXiv:physics*/0403033 (2004). *Get Preprint*

✱25 C. Kauffmann in A. van der Burgh (ed.). *Topics in Engineering Mathematics*, Kluwer, Dordrecht, 1993. *BookLink*

✱26 R. Khanin in B. Mourrain (ed.). *ISSAC 2001*, ACM, Baltimore, 2001. *DOI-Link*

✱27 P. Kleinert, H. Schlegel. *Physica* A 218, 507 (1995). *DOI-Link*

✱28 D. E. Knuth. *Am. Math. Monthly* 81, 323 (1974).

✱29 D. E. Knuth. *Am. Math. Monthly* 92, 170 (1985).

✱30 G. Kuperberg. *arXiv:math.HO*/0210144 (2002). *Get Preprint*

✱31 J. D. Logan. *Applied Mathematics*, Wiley, New York, 1987. *BookLink (2)*

✱32 K. C. Louden. *Programming Languages: Principles and Practice*, PWS-Kent, Boston, 1993. *BookLink (2)*

✱33 R. Maeder. *Programming in Mathematica*, Addison-Wesley, Reading, 1997. *BookLink (3)*

✱34 R. Maeder. *The Mathematica Programmer*, Academic Press, New York, 1993. *BookLink (2)*

✱35 B. S. Massey. *Measures in Science and Engineering*, Wiley, New York, 1986. *BookLink*

✱36 G. Messina, S. Santangelo, A. Paoletti, A. Tucciarone. *Nuov. Cim.* D 17, 523 (1995).

✱37 J. Molenaar in A. van der Burgh, J. Simonis (eds.). *Topics in Engineering Mathematics*, Kluwer, Dordrecht, 1992. *BookLink*

✱38 E. Pascal. *Repertorium der höheren Mathematik* Theil 1/1 [page V, paragraph 3], Teubner, Leipzig, 1900.

✱39 S. H. Romer. *Am. J. Phys.* 67, 13 (1999). *DOI-Link*

✱40 R. Sedgewick, P. Flajolet. *Analysis of Algorithms*, Addison-Wesley, Reading, 1996. *BookLink*

✱41 R. Sethi. *Programming Languages: Concepts and Constructions*, Addison-Wesley, New York, 1989. *BookLink*

✱42 D. B. Wagner. *Power Programming with Mathematica: The Kernel*, McGraw-Hill, New York, 1996. *BookLink*

✱43 S. Warner. *arXiv:cs.DL*/0101027 (2001). *Get Preprint*

✱44 H. Whitney. *Am. Math. Monthly* 75, 115, 227 (1968).

✱45 S. Wolfram. *The Mathematica Book*, Wolfram Media, Champaign, 2003. *BookLink*

P R O G R A M M I N G

CHAPTER 1

Introduction to *Mathematica*

1.0 Remarks

In this first chapter, we give a general overview of the abilities and possible applications of *Mathematica* by examples, along with some of its limitations. We present the most important syntactic differences between *Mathematica* and other programming languages, including the use of symbols, parentheses `()`, braces `{}`, and brackets `[]`. The preferred way for formatting source code is also discussed. A short tour is taken through the numerical, graphical, symbolic, and programming capabilities of *Mathematica*. One important subject omitted (because the main focus of this book series is the application of *Mathematica* to problems from the natural sciences and engineering) is the typesetting- and electronic document-related feature set of *Mathematica*. See *The Mathematica Book* [1382✱] and [538✱] for details.

Some of the inputs shown and executed in this chapter represent an intermediate to advanced use of *Mathematica*. Readers new to *Mathematica* will probably not understand how they work, neither should they. These inputs and code pieces represent a cross section of the type of problems treated in this book. After reading the *GuideBooks*, the reader will have no problem understanding these programs.

All notebooks will have the following initialization cell. It will turn off possible spelling error messages, set the default fonts and font sizes for labels in graphics, and reset the line numbering such that evaluating a section or a subsection will start with In[*n*].

```
(* no spelling warnings, set fonts for tick labels, ... *)
Get[ToFileName[ReplacePart["FileName" /.
 NotebookInformation[EvaluationNotebook[]], "Initialization.m", 2]]];
```

1.1 Basics of Mathematica as a Programming Language

■ 1.1.1 General Background

Mathematica is an interactive programming system. To begin programming in *Mathematica*, start the *Mathematica* application. (The *Mathematica* kernel can also be run in batch mode. On a UNIX system, type `(time math <` *inputFileName*`) >!` *outputFileName* `&` at the prompt to run the kernel in batch mode.)

The following example shows the first input and output lines of an initial *Mathematica* session [611✱]. (In[*n*]:= and Out[*n*]= are generated by *Mathematica*, and not input by the user.)

```
1 + 1
```

```
(-2) * (-2)
```

Everything done to this point in a given *Mathematica* session is saved in the values of the variables `In` and `Out`.

The following list provides the basic rules for the use of *Mathematica* as a programming language.

■ Almost all built-in commands (we will use the words "command" and "function" interchangeably in the *GuideBooks*) begin with a capital letter and are nonabbreviated, standard English words. If a command consists of several words, the first letter of each word comprising the command is also a capital letter. The complete word is written without spaces, (e.g., `AxesLabel`, `ContourSmoothing`, and `TeXForm`). If the name of a person is involved, for example, in the special functions of mathematical physics, the name comes first, followed immediately by the usual symbol for this function, represented by a capital letter (e.g., `JacobiP`, `HermiteH`, `BesselJ`, and `RiemannSiegelZeta`).

Two classes of exceptions exist to this general rule. The first class concerns mathematical notation: Shorter symbols are used—such as `E` for the number e, `I` for $i = \sqrt{-1}$, `Det` for determinant, `Sin` for sine, and `LCM` for the least common multiple. The second class includes the abbreviation `N` for numerical operations (e.g., `N` for the computation of numerical values themselves, such as `N[Sqrt[2]]`, which evaluates and prints as 1.41421); and `NSolve` for the numerical solution of equations); the abbreviation `D` for operations involving differentiation (e.g., `D` for differentiation and `DSolve` for solving differential equations); and the abbreviation `Q` (question) for functions asking questions (e.g., `EvenQ` for testing if something is an even number). *Mathematica* knows about one thousand executable commands.

■ Symbols defined by the user usually begin with lowercase letters. Variable names can be arbitrarily long and include both uppercase and lowercase letters, `$`, and numbers (but numbers cannot be used as the first character). Only complete, well-developed routines should be given names starting with capital letters (as mentioned in the preface, we will not strictly follow this convention). Names of the form *name1_name2* are not allowed in *Mathematica* (one can input an expression of the form *name1_name2*, but *Mathematica* does not interpret this as one name). Users should never introduce symbols of the form *name*`$` or *name*`$`*number* because *Mathematica* produces symbols in this form to make names unique (see Chapter 4).

■ The operation of many *Mathematica* functions can be influenced by a variety of options of the form *optionName* `->` *specialOptionSetting* (e.g., `PlotPoints -> 25` and `Method -> GaussKronrod`). The possible settings for the options of a command depend on the command and include numbers, lists, or such things as `All`, `None`, `Automatic`, `True`, `False`, `Bottom`, `Top`, `Left`, `GaussKronrod`, and `CofactorExpansion`. Around 450 differently named options exist. For simple options, these names are *Mathematica* expressions, for more specialized options they are typically strings. Options can sometimes contain suboption settings.

■ About 120 commands work together with *Mathematica* as a general programming (computer-dependent) system and begin with `$` (e.g., `$MachineEpsilon` and `$MachineType`).

■ Mathematical functions rarely used, or used only for special purposes, are not implemented in the kernel, which is written in C. They are often available in external packages, which are written in *Mathematica*. To use these functions, one must first load the appropriate package. The same naming conventions apply. For operating systems allowing arbitrarily long file names, these packages have names of the form *Subject*`` ` ``*SpecialTopic*`` ` `` (e.g., ``Algebra`Quaternions` ``, ``DiscreteMath`CombinatoricalFunctions` ``, and ``NumericalMath`BesselZeros` ``) and are loaded using `Needs["`*Subject*`` ` ``*SpecialTopic*`` ` ``"]` or `Get["`*Subject*`` ` ``*SpecialTopic*`` ` ``"]`.

■ Error messages of the form `command::`*nameOfTheError*`:`*RoughSpecificationOfTheError* result when syntactically incorrect source code is input, the wrong number of arguments is given, the wrong type of argument is given for a particular command, or errors arise in the calculation. For example, the input

```
Plot[Sin[x], {x, 0, soFarThatANicePictureComesOut}]
```

produces the following message:

```
Plot::plln: Limiting value soFarThatANicePictureComesOut in
{x, 0, soFarThatANicePictureComesOut} is not a machine-size real number.
```

Σ (* session summary *) **`TMGBs`PrintSessionSummary[]`**

■ 1.1.2 Elementary Syntax

The algebraic operations addition, subtraction, multiplication, and division are denoted as usual by `+`, `-`, `*`, and `/`. The `*` for multiplication can be omitted by using a blank space instead. Parentheses `()` are used exclusively for grouping, and brackets `[]` are used for enclosing arguments in functions. Braces `{}` are used to enclose components of vectors and elements of sets (here, any number of elements of arbitrary type are allowed, which can be nested to any level).

Mathematical Expression		***Mathematica Form***
Addition $c+b$	⟶	`c + b`
Subtraction $d-e$	⟶	`d - e`
Multiplication $3\,x$	⟶	`3 x` or `3*x`
Division $4/r$	⟶	`4/y` (`4/x y` is `(4/x)*y`)
Exponentiation h^l	⟶	`h^l`
Grouping $(2+3)\,4$	⟶	`(2 + 3) 4`
Function with an argument $f(x)$	⟶	`f[x]`
Discrete iterator $i = 1, 2, 3, \ldots, 9, 10$	⟶	`{i, 1, 10, 1}` or `{i, 10}`
Continuous range $x = 0 \ldots 1$	⟶	`{x, 0, 1}`
Vector $\{a_x, a_y, a_z\}$	⟶	`{ax, ay, az}`
Decimal number 3.567	⟶	`3.567`
Assignment $x = 3$	⟶	`x = 3`
Mathematical equality $\sin(\pi/2) = 1$	⟶	`Sin[Pi/2] == 1`
Function definition $f(x) = \sin(x)$	⟶	`f[x_] := Sin[x]`
String "hello world"	⟶	`"hello world"`
"Collection" of items $\{apple, apple, \mathbb{Z}\}$	⟶	{*apple*, *apple*, ℤ}

The following list describes the syntax used in *Mathematica*:

■ The *i*th element of (a_x, a_y, a_z): `{ax, ay, az}[[`*i*`]]` (*i* is a concrete positive integer number)

■ Prevent the display of (long) results by using a semicolon at the end of input: *expression*`;`

■ The last expression given by *Mathematica*: `%`

■ The next-to-last (penultimate) expression given by *Mathematica*: `%%`

■ The *i*th output of *Mathematica*: `%`*i* or `Out[`*i*`]`

■ When an expression is too long to fit on one line, the symbol \ (or ⋱) is displayed, indicating that the expression is continued on the next line (if an expression is incomplete when the end of the line is reached, the expression is automatically considered to be continued on the next line)

■ Comments can be written in the form, `(* material to be ignored when sent to the` *`Mathematica`* `kernel *)` (comments can be inserted anywhere in *Mathematica* source code)

■ Information on the command *command*: `?`*command*

■ More information on the command *command*: `??`*command*

■ Metacharacter inside a string (standing for an arbitrary symbol): `*`

■ Options of functions are set in the form *option* `->` *value*, for instance: `PlotPoints -> 50`.

■ "Ordinary", Greek, Gothic, Script, and doublestruck letters represent different letters (B≠Β≠𝔅≠ℬ≠𝔹), and symbol names made from them are considered different. But plain, bold, italic, bold-italic, and underlined versions of a letter are considered equal (B=**B**=*B*=***B***=B). (The *Mathematica* inputs of the *GuideBooks* will make use of "ordinary", Greek, Gothic, Script, and doublestruck letters, but all inputs will be in bold-nonitalic.) As the default output format, we will use `StandardForm`. In `StandardForm`, some symbols appear in a slightly "doubled" version. Most frequently, we will encounter ⅇ for *e*, the base of the natural logarithm, ⅈ for $\sqrt{-1}$, and ⅆ for the differential *d* in integrals.

■ Independent inputs can either be placed on separate lines or they can be separated by semicolons: *inputStatement*$_1$`;` *inputStatement*$_2$`;` …`;` *inputStatement*$_n$.

The use of parentheses `(`*someExpressions*`)` for grouping and brackets `[`*argumentsOfAFunction*`]` for arguments of functions is essential for correct syntax; braces `{}` and double square brackets `[[`*sequenceOfPositiveIntegersOr0*`]]` are short forms for the commands `List` and `Part`.

Using a functional programming style, it is often possible to write *Mathematica* code without using auxiliary variables. As a consequence, a large number of brackets `[]` is often needed. In order to make such parts of a program easier to understand, the convention used (if space allows) in this book series is to align corresponding pairs of brackets `[...]` and often pairs of `()` and `{}` vertically or horizontally (but this is a matter of the user's personal taste). This process usually means indenting the code appropriately. Thus, *Mathematica* source code for programs should be printed using families of monospaced fonts with equally sized letters, such as `Courier`. It is common to include blank spaces around relatively weak operators, such as `+`, `_`, or `->`. This convention does not apply inside short forms of commands. Sixty-five commands in *Mathematica* have short forms; around 50 of these commands consist of two or three ASCII characters (e.g., `->` [`Rule`] for replacement, `!=` [`Unequal`] for inequality). No blank spaces are allowed between the symbols in these short forms. Relatively short *Mathematica* inputs representing mathematical expressions often look better in `StandardForm` notation (in `StandardForm` no additional spaces should be added). Because this book contains a lot of code and to maintain uniformity, we will use `InputForm` throughout this book. In some rare cases, we will use `StandardForm`, mainly for demonstration purposes.

In procedural programs, we will typically use one line per procedural statement. If possible and appropriate, we will carry out multiple assignments at once (for instance `{one, two} = {1, 2}` instead of `one = 1; two = 2`).

Below is an example of the general rules for *Mathematica* source code. In addition to the formatting, note that named temporary auxiliary variables can be largely dispensed with using *Mathematica*'s functional programming capabilities. In the following code only, the variables `armed`, `numberOfPoints`, and `rotation` in the function definition appear; no further user-defined variables exist. Starting from now, we will display user-changeable arguments in italic. For the function `RotatedBlackWhiteStrips` below the three arguments *armed*, *numberOfPoints*, and *rotation* are user-changeable arguments. The frequent appearance of `#` and `&` are parts of so-called pure functions; we discuss them in detail in Chapter 3.

It is a common convention in *Mathematica* that, whenever possible, a "typical" mathematical symbol (character sequence) for a quantity should be used. If not, a notation should be chosen to reflect the effect of the corresponding command or the contents of the corresponding list.

Readers will probably not understand the following code initially. However, after reading this book and looking at this code again, they will have no problem understanding how it works.

```
RotatedBlackWhiteStrips[
            armed_Integer?((# >= 4 && EvenQ[#])&),
            numberOfPoints_Integer?(# > 3&), rotation_?(Im[#] == 0&)] :=
Graphics[ (* black or white? *)
MapIndexed[{If[(-1)^Total[#2] == 1,
               GrayLevel[0], GrayLevel[0.8]],
       (* make polygons *)
       Polygon[Join[#1[[1]], Reverse[#1[[2]]]]]}&,
            Partition[
              Partition[(* calculate vertices *)
              Distribute[{N[{{+Cos[#], Sin[#]},
                             {-Sin[#], Cos[#]}}]& /@
                               Range[0, 2Pi, 2Pi/armed],
                         N[( 1 - (#/(2Pi)))*
                            {Cos[rotation #], Sin[rotation #]}
                           ]& /@ Range[0, 2Pi, 2Pi/numberOfPoints]
                         }, List, List, List, Dot],
            numberOfPoints + 1],
             {2, 2}, 1],
     {2}], (* options for a nice-looking graphic *)
          AspectRatio -> Automatic, PlotRange -> All]
```

We now look at three short examples of `RotatedBlackWhiteStrips` [762✱], [489✱].

```
Show[GraphicsArray[{RotatedBlackWhiteStrips[ 4, 24,  1/4],
                    RotatedBlackWhiteStrips[12, 36, -1/8],
                    RotatedBlackWhiteStrips[72, 36,  1/4]}]]
```

In the programming code, we will try adhere to the aforementioned formatting conventions. But because of both horizontal and vertical space limitations on the pages of the book, it will not always be possible to follow the conventions exactly in every piece of code. Closing parentheses, brackets, and braces will not often be aligned vertically with the corresponding opening ones. Successive arguments of functions will either be written in one line or sometimes aligned vertically. This is in particular the case when a program uses many nested (pure) functions such as following. Here we partition a regular *n*-gon (*n* even) into rhombuses (once again, we make no use of temporary auxiliary variables).

```
GrayRhombusPartition[n_?(EvenQ[#] && # > 4&), opts___] :=
Graphics[ (* make gray colors *)
{MapIndexed[{GrayLevel[(#2[[1]] - 1)/(n/2 - 2)], #1}&,
            MapThread[Polygon[ (* make polygons *)
                         Join[#1, Reverse[#2]]]&, #]& /@
   ((Partition[#, 3, 2]& /@ #&) /@
      ({Drop[Drop[#[[1]], 1], -1], #[[2]]}& /@
          Partition[#, 2, 1]))],
(* make lines *)
{Thickness[0.15/n],
 MapIndexed[{GrayLevel[1 - (#2[[1]] - 1)/(n/2 - 1)], #1}&,
             Line /@ #]}}&[
(* the points calculated by iteration *)
Drop[Flatten[Transpose[{#1, Join[#2, {{}}]}], 1], -1]& @@ #& /@
NestList[{Last[#],
          2(Total[#]/2& /@ Partition[Last[#], 2, 1]) -
          Drop[Drop[First[#], 1], -1]}&,
         N[{Array[{0, 0}&, {n/2 + 1}],
            Array[{Cos[Pi/n(1 + 2#)], Sin[Pi/n(1 + 2#)]}&, n/2, 0]}],
         n/2 - 1]], AspectRatio -> Automatic, opts]
```

Here are two examples using `GrayRhombusPartition`.

```
Show[GraphicsArray[
{GrayRhombusPartition[ 8, Background -> Hue[0.12]],
 GrayRhombusPartition[28, Background -> Hue[0.12]]}]]
```

Obeying strictly the above-formulated guidelines, this routine is quite big and nearly “ununderstandable” if formatted “properly” on paper.

```
GrayRhombusPartition[n_?(EvenQ[#] && # > 4&), opts___] :=
Graphics[
 Function[      (* ≈ 100 lines deleted for brevity  *)
         ][     (* ≈ another 120 lines deleted for brevity  *)
          ], Rule[AspectRatio, Automatic]
        ]
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

1.2 Introductory Examples

1.2.0 Remarks

In this section, we will give a short overview of the mathematical, graphical, and numerical possibilities built into *Mathematica*. The examples are largely unrelated to each other. We discuss all graphics-related commands in the Graphics volume of the *GuideBooks* [1283✱] and mathematics-related *Mathematica* commands in detail in the Numerics [1284✱] and Symbolics [1285✱] volumes. *Mathematica* also contains a fully developed programming language. We will discuss programming-related features in detail in the next five chapters. The meaning of some of the inputs will be clear to readers without prior *Mathematica* experience. Some of the inputs will use commands that are not immediately recognizable; others will use “cryptic” shortcuts. In the following chapters, we will discuss the meaning of all the commands, as well as their aliases, in detail.

The division into programming, graphics, numerics, and symbolics does not reflect the structure of *Mathematica*. Just the opposite: The harmonic and fluent connection between all functions makes *Mathematica* an integrated environment where all parts can be used together in a smooth way. Also, the division into numerics and symbolics is not a strict one: To derive efficient numerical methods, one needs symbolic techniques, and for carrying out complicated symbolic calculations, one frequently needs validated numeric decision procedures.

The examples of this chapter form a “random” collection. By no means are they intended to give up a complete, coherent, and logically built overview of *Mathematica*. Its capabilities are much too many and too diverse to even try to give such an overview inside one chapter.

1.2.1 Numerical Computations

$\sin(\pi/3)$ gives an “exact number”.

```
Sin[Pi/3]
```

We can compute this number to machine accuracy. Six digits are usually displayed.

```
N[Sin[Pi/3]]
```

Here are 18 digits in the result.

```
N[Sin[Pi/3], 18]
```

We can also compute and display a result with 180 digits.

```
N[Sin[Pi/3], 180]
```

This input calculates the first 1000 terms of the simple continued fraction expansion of $\sqrt[3]{5}$. (As mentioned above, the semicolon at the end of an input avoids that the result is printed.)

```
cf = ContinuedFraction[5^(1/3), 1000];
```

This result shows the number of times various integers appear in the continued fraction expansion.

```
Map[(* count occurrences *) Function[digit, {digit, Count[cf, digit]}],
     (* all occurring integers *) Union[cf]]
```

The next input counts the number of occurrences of the number 1 in the first million continued fraction digits of $\sqrt[3]{5}$. This can be done in a few seconds.

```
Count[ContinuedFraction[5^(1/3), 1000000], 1] // Timing
```

Continued fractions of square roots are ultimately periodic.

```
ContinuedFraction[66^(1/2), 20]
```

Is $\sqrt[3]{\exp(\pi\sqrt{163}) - 744}$ the integer 640320? The answer is no, but it "almost" is [1249✱].

```
Element[(E^(Sqrt[163] Pi) - 744)^(1/3), Integers]

N[(E^(Sqrt[163] Pi) - 744)^(1/3) - 640320, 60]
```

To find out that $\sqrt[3]{\exp(\pi\sqrt{163}) - 744}$ is less than 640320, one does not have to use explicitly a numerical approximation. Just evaluating the comparison $\sqrt[3]{\exp(\pi\sqrt{163}) - 744} < 640\,320$ causes *Mathematica* to carry out all necessary calculations to answer this question.

```
(E^(Sqrt[163] Pi) - 744)^(1/3) < 640320
```

For an explanation of why this number is almost an integer, see [294✱], [1338✱], and [1141✱]; for similar identities, see [922✱].

Much more extreme cases exist of numbers that are almost integers. They are called Pisot numbers (see [138✱], [139✱], [711✱], [408✱], [868✱], [185✱], and [887✱]). Consider

$$\left(\frac{\sqrt[3]{2}}{\sqrt[3]{27 + 3\sqrt{69}}} + \frac{\sqrt[3]{27 + 3\sqrt{69}}}{3\sqrt[3]{2}}\right)^{27\,369}.$$

The result is not an integer, but it nearly is.

```
N[(2^(1/3)/(27 + 3 Sqrt[69])^(1/3) +
   (27 + 3 Sqrt[69])^(1/3)/(3 2^(1/3)))^27369,
   (* numericalize to 5030 digits *)  5030
   (* why 5030?
  $MaxExtraPrecision = Infinity;
  (* left of decimal point  3342 *) Floor[Log[10, Round[#]] -
  (* right of decimal point 1671 *) Floor[Log[10, # - Round[#]]] +
  (* some more non-9-digits 0017 *)]&[theRadical] *)]  -
248872083860566242801488633985778816168566582615463984666186327177996889794
302876969944745816129045615885143011927101923791713997993058914014883941331
965886658596361798867563654794840763150485611020414502205710144974280728374
349044713489229346181918805096874878013575556923353742673696224778320245988
540213301883484666470466149889402655143734621040204402439497074243583844435
808572284035809706292967988993338265986862439878547167243747603358101005823
770325288671140498237982079089990431287680958041449065611648473793797460006
542685289106532890742345783983687027507936729079442473934078360160815378816
494153662235479538964578833871970301073249242325586046493271959208073441641
940884995001297965439527338534109556225631472247772230281824440018654558291
013684116069229948450508385560502376379491505913877574694543067098950233734
875259586944931660657861461142958051706161345801562687419677892445722586732
551348551144898211307412861644702494277043219675492384705090308683393258398
456210775092840495926289398412204946622896060874294857076651762085967637510
077537670566013460187710270680862338508370476316341613384164718123490256852
014554906330744898465446950034570811433400237285702426141033340407021679373
889901563587912181986503488932240588333472792264516219643268144193209629883
670458727361899797093663301089446836229230254803886092708925799050583760656
437272267338242109959665203275242365597028650587908842357311629984324872399
370681856106228825253081951335763606805097314767756009989894248180226689216
887125546603079786776420339174335241777036123462355674280571688628682637154
744918786523022395903717847865060788592985252403020060537542636129564913749
579902728693786036767203892699418847034973900792486513050707875184722293046
683552341178497622788847536427384240325375931710068928003083282083508259416
757110641854633899165463352000712509400393706057751324434941912458367864031
438044741715469307650984762987113625655095113341410659514797573216487308588
207929723616047980118369534484150697741703276041764283828990373663679698758
383036224461356559232344645741738783654670759079114885744233509780436530814
758237796222541372347526347511124157083242577253654864546653468558226069365
215604513857702802435076942062477624009724087750511435288253440943800323682
814500906873898893269944000616164741243202139992999892419706344951703777826
055705878691043258271291941546764790768702904202815388755953467402952252786
242105372182173621873752243352251007748639891006060850310559871809504335746
400950552625647975671614005288806192143795350726970553183450775224485377787
848075149669430514248120843405866305425664958833381695289311873275612903811
625316839963397212327107969696245976920848255222591348999445674453161441801
149262472389961197753334548229672385129687618298798276361290308183040642825
761789360866674785134042824865250319983289744838888137526494195021927158720
980424579870985098762439838255243931303193820158912431012986549938720840348
650585370461953198199414358447110283006585773942850787801658598482880852634
887038330953482823346606566055339838200632031259942468414620516606902878898
959050373271686613923208614965923844927939159262755102043035136468782747102
192779859301117801065439219569499299420368424993003990461640112615325982631
808971152916585811064172283699654029309129460623214205826005262694547534088
```

The last output has 1670 consecutive 9s.

```
StringLength[First[StringCases[ToString[%], "9" ..]]]
```

Infinitely many such numbers exists, whose high powers are almost integers.

Arithmetic operations with integers always lead to exact results.

```
111^111
```

Even 1111^{1111} can be computed in a (nearly) vanishing amount of time. (We use `Short` to suppress printing the entire number and give only some of its first and last digits.)

```
Short[Timing[1111^1111] // OutputForm]
```

Mathematica can deal quickly with large integers. Here are two integers, both having more than one million digits.

```
int1 = 111111^222222;
int2 = 222222^333333;
```

```
N[{int1, int2}]
```

Multiplying these two integers can be done in a few seconds on a modern computer. (The actual calculation was carried out on a 2 GHz computer.)

```
Timing[int3 = int1 int2;]
```

The resulting number has more than 2.9 million digits.

```
N[int3]
```

Here is the total number of digits in base 2—nearly ten million digits.

```
Length[IntegerDigits[int3, 2]]
```

But the reader should keep in mind that *Mathematica* is an interpreted language. It does not carry out any meaning- and/or result-changing optimization automatically. So the following simple loop takes a few seconds.

```
Do[1, {10^8}] // Timing
```

The following picture shows the distribution of the digitsums of 1000 random integers between 1 and 10^{10}. Each color represents the digitsum in base b, where $2 \leq b \leq 50$.

```
With[{(* 1000 random numbers *)
      randomNumberList = Table[Random[Integer, {1, 10^10}], {1000}]},
     Show[Graphics[{PointSize[0.005],
                    (* different color for each base *)
                    MapIndexed[{Hue[#2[[1]]/60], #}&,
                     MapIndexed[Point[{#2[[2]], #1}]&,
                         (* digitsums of the random numbers *)
                         Table[Sort[Total[IntegerDigits[#, b]]& /@
                                                 randomNumberList],
                               {b, 2, 50}], {2}]]}],
          PlotRange -> All, Frame -> True]]
```

Here is a simple numerical integration: $\int_0^1 x^3\, dx$.

```
NIntegrate[x^3, {x, 0, 1}]
```

In the following numerical integration $\int_0^1 1 \big/ \sqrt{x}\; dx$, the function is integrable, but it has a singularity at $x = 0$.

```
NIntegrate[1/Sqrt[x], {x, 0, 1}]
```

Here is a contour integral in the complex z-plane (by the Residue theorem, its value is $2\,\pi\, i$).

$$\int_1^i \frac{1}{z}\, dz + \int_i^{-1} \frac{1}{z}\, dz + \int_{-1}^{-i} \frac{1}{z}\, dz + \int_{-i}^{1} \frac{1}{z}\, dz$$

```
NIntegrate[1/z, {z, 1, I, -1, -I, 1}]
```

The small, real part comes from the use of an approximating method and approximate numbers. Using *Mathematica*'s high-precision arithmetic, we can get more correct digits.

```
NIntegrate[1/z, {z, 1, I, -1, -I, 1}, WorkingPrecision -> 50]
```

Of course, *Mathematica* can carry out this integral also exactly.

```
Integrate[1/z, {z, 1, I, -1, -I, 1}]
```

Next, we numerically solve the differential equation: $x''(t) + x'(t)^3 / 20 + x(t)/5 = \cos(e\,t)/3$, with initial conditions $x(0) = 1$ and $x'(0) = 0$ (a forced nonlinear oscillator with damping [703✱], [587✱]).

```
sol = NDSolve[{(* differential equation *)
               x''[t] + 1/20 x'[t]^3 + 1/5 x[t] ==  1/3 Cos[E t],
               (* initial conditions *)
               x[0] == 1, x'[0] == 0}, x[t], {t, 0, 360}]
```

The result is an approximate solution represented in *Mathematica* as an `InterpolatingFunction`-object that is embedded in a replacement rule `{x -> `*solution*`}`. We can now plot it.

```
Plot[Evaluate[x[t] /. sol], {t, 0, 100}]
```

The next picture shows a phase-portrait of the oscillations.

```
ParametricPlot[Evaluate[{x[t] /. sol[[1]], D[x[t] /. sol[[1]], t]}],
               {t, 0, 360},
               Frame -> True, Axes -> False, PlotPoints -> 3600]
```

Here is a more complicated system of differential equations—the so-called Burridge–Knopoff model for earthquakes [217✱], [460✱], [976✱], [1080✱], [372✱], [1352✱], [371✱], [598✱], [1397✱]. n points $x_i(t)$ on a straight line, each of mass m interact with each other via springs of stiffness k_c, all masses are subject to a force that is proportional to the distance of the masses from their equilibrium position and to a friction force $\mathcal{F}(v)$. (The equilibrium position of mass i is $i\,a$.)

$$m\,x_i''(t) = k_c\,(x_{i-1}(t) - 2\,x_i(t) + x_{i+1}(t)) - k_p\,(x_i(t) - i\,a - t\,v) - f\,\mathcal{F}\big(f\,x_i'(t)\big)$$

```
odeSystem[n_, {m_, kc_, kp_, v_, a_, f_, 𝑓_}] :=
 Table[m x[i]''[t] ==  kc (x[i + 1][t] - 2 x[i][t] + x[i - 1][t])
                      -kp (x[i][t] - i a - v t) - f 𝓕[𝑓 x[i]'[t]],
       {i, n}] /. (* remove first and last masses *)
                 {x[0][t] :> x[1][t] - 1, x[n + 1][t] :> x[n][t] + 1}
```

We choose $\operatorname{sgn}(v)\,|v|^{1/2}\,e^{-|v|}$ for $\mathcal{F}(v)$.

```
𝓕[v_?NumberQ] := Sign[v] Sqrt[Abs[v]] Exp[-Abs[v]];
```

The function `solveODEsAndShowSolutions` solves the system of equations for given values of the parameters under certain initial conditions.

```
solveODEsAndShowSolutions[{n_, T_},
                          {m_, kc_, kp_, v_, a_, f_, 𝑓_}, opts___] :=
Module[{nsol},
(* solve differential equations *)
nsol = NDSolve[Flatten[{odeSystem[n, {m, kc, kp, v, a, f, 𝑓}],
(* initial conditions *)
Flatten[Table[{x[i][0] == i a + a Cos[i]/3, x[i]'[0] == 0}, {i, n}]]}],
            Table[x[i], {i, n}], {t, 0, T}, opts,
            MaxSteps -> 10^5, PrecisionGoal -> 5, AccuracyGoal -> 5];
(* display solutions *)
Plot[Evaluate[Table[x[i][t] - v t, {i, n}] /. nsol], {t, 0, T},
     PlotRange -> All, PlotStyle -> {Thickness[0.002]},
     Frame -> True, Axes -> False, PlotPoints -> 500]]
```

Here is the solution for a numerical set of parameters shown. One clearly sees collective motions of the particles caused by their nonlinear coupling.

```
solveODEsAndShowSolutions[{50, 50}, (* parameter values *)
                {-0.826801, -8.710866, -0.195864, -0.709007,
                 -9.852322,  1.596424, -3.359798}]
```

Next, we consider a particle in a two-dimensional potential that has confining quadratic part and a random, smoothly oscillating part:

$$V(x,\,y) = x^2 + y^2 + \sum_{i,j=0}^{o} r_{i,j}\cos\left(i\,x + \varphi_{i,j}^{(x)}\right)\cos\left(j\,y + \varphi_{i,j}^{(y)}\right).$$

Here the $r_{i,j}$ are random variables from the interval $[-1,\,1]$ and the $\varphi_{i,j}^{(x)}$, $\varphi_{i,j}^{(y)}$ random phases from the interval $[0,\,2\,\pi]$. We assume frictionless motion and solve the equations of motions; four coupled nonlinear ordinary differential equations of first order, for a time T. Instead of explicitly specifying the $3\,(o+1)^2$ random parameters, we seed the random number generator using a 20-digit seed *seed*.

The code for `random2DPotentialParticlePath` is longer than the above inputs because, in addition to solving the equations of motions and plotting the particle path, we color the path according to the particle's velocity (red being slow and blue being fast), show the zero-velocity contour as a guide for the eye of the reachable configuration space, and show the potential itself as a contour plot underneath.

```
random2DPotentialParticlePath[o_, seed_, T_, pp_, opts___] :=
Module[{V, x, y, vx0, vy0, nsol, pathData, path, xMin, xMax,
        yMin, yMax, zeroVelocityContour, potentialLandscape},
  (* seed random number generator *) SeedRandom[seed];
  (* generate random potential *)
  V[x_, y_] = x^2 + y^2 + (* random part of the potential *)
             Sum[Random[Real, {-1, 1}]*
                 Cos[i x + 2Pi Random[]] Cos[j y + 2Pi Random[]],
                 {i, 0, o - 1}, {j, 0, o - 1}];
  (* random initial velocity components *)
  {vx0, vy0} = Table[Random[Real, {-2, 2}], {2}];
  (* solve Newton's equations *)
  nsol = NDSolve[{x'[t] ==  vx[t], y'[t] == vy[t],
                 vx'[t] == -D[V[x[t], y[t]], x[t]],
                 vy'[t] == -D[V[x[t], y[t]], y[t]],
                 (* initial conditions *)
                 x[0] == 0, y[0] == 0, vx[0] == vx0, vy[0] == vy0},
                 {x, y, vx, vy}, {t, 0, T}, MaxSteps -> 10^5];
  (* position and velocity data *)
  pathData = Table[Evaluate[{{x[t], y[t]}, {vx[t], vy[t]}} /.
                                   nsol[[1]]], {t, 0, T, T/pp}];
  (* particle path; colored according to velocity *)
  path = {Hue[0.5 ArcTan[Sqrt[#.#]&[(#1[[2]] + #2[[2]])/2]]],
         Line[{#1[[1]], #2[[1]]}]}& @@@ Partition[pathData, 2, 1];
  (* maximal x,y-extensions *)
  {{xMin, xMax}, {yMin, yMax}} = {#1 - #3/12, #2 + #3/12}&[
    Min[#], Max[#], Max[#] - Min[#]]& /@ Transpose[First /@ pathData];
(* zero-velocity contour and contour plot of the potential *)
  {zeroVelocityContour,  potentialLandscape} =
  ContourPlot[Evaluate[V[x, y]], {x, xMin, xMax}, {y, yMin, yMax},
             DisplayFunction -> Identity,  PlotPoints -> 240, ##]& @@@
   (* set options for contour plot *)
   {{Contours -> {(vx0^2 + vy0^2)/2 + V[0, 0]}, ContourShading -> False,
     ContourStyle -> {{GrayLevel[0], Thickness[0.002]}}},
    {Contours -> 100, ColorFunction -> (GrayLevel[1 - #]&),
     PlotRange -> All, ContourLines -> False}};
  (* show potential, zero-velocity contour, and particle path *)
  Show[{potentialLandscape, zeroVelocityContour,
       Graphics[{Thickness[0.002], path}]}, opts,
       AspectRatio -> 1, PlotRange -> All, Frame -> False,
       DisplayFunction -> $DisplayFunction]]
```

Here are three example potentials and particle paths for $o = 3$, $o = 12$, and $o = 10$. The first two motions are pseudoperiodic. The third motion is chaotic and the particle samples the accessible configuration space in a complicated manner [704✱]. The potential used in the last graphic has $3 \times 11^2 = 363$ random parameters.

```
Show[GraphicsArray[
Block[{$DisplayFunction = Identity},
 random2DPotentialParticlePath[##, 10 #3]& @@@
                 (* pseudoperiodic motion *)
                 {{ 3, 21598974805925082378, 200},
                  {12, 60923097090049506424,  50},
                 (* chaotic motion *)
                  {10, 58211857412104937056, 200}}]]]
```

Mathematica can also solve partial differential equations. Here is the so-called Benney equation in 1 + 1 dimensions [1148✱], [1059✱], a nonlinear partial differential equation.

$$\frac{\partial\psi(x,t)}{\partial t}+\psi(x,t)\,\frac{\partial\psi(x,t)}{\partial x}+\frac{\partial^2\psi(x,t)}{\partial x^2}+\varepsilon\,\frac{\partial^3\psi(x,t)}{\partial x^3}+\frac{\partial^4\psi(x,t)}{\partial x^4}=0$$

We will solve the Benney equation for $\varepsilon = 0.001167$, periodic boundary conditions, and the following initial condition (a “random”, oscillating function of magnitude $\simeq 10^0$):

$$\psi(x,0)=\frac{1}{25}\cos\left(\frac{\pi x}{20}\right)-\frac{38}{191}\cos\left(\frac{\pi x}{25}\right)-\frac{11}{116}\cos\left(\frac{\pi x}{40}\right)-\frac{21}{23}\cos\left(\frac{\pi x}{50}\right)+\frac{79}{140}\cos\left(\frac{3\pi x}{50}\right)+$$
$$\frac{7}{55}\cos\left(\frac{\pi x}{100}\right)-\frac{4}{131}\cos\left(\frac{3\pi x}{100}\right)-\frac{95}{101}\cos\left(\frac{\pi x}{200}\right)-\frac{115}{166}\cos\left(\frac{3\pi x}{200}\right)-$$
$$\frac{3}{5}\cos\left(\frac{7\pi x}{200}\right)-\frac{9}{16}\cos\left(\frac{9\pi x}{200}\right)-\frac{100}{123}\cos\left(\frac{11\pi x}{200}\right)+\frac{12}{19}\cos\left(\frac{13\pi x}{200}\right).$$

Solving a partial differential equation is more time-consuming than solving an ordinary differential equation; the following inputs need longer than the above one to complete.

```
(* the differential equation *)
pde = D[ψ[x, t], t] + ψ[x, t] D[ψ[x, t], x] + D[ψ[x, t], {x, 2}] +
      ε D[ψ[x, t], {x, 3}] + D[ψ[x, t], {x, 4}];

(* the initial condition *)
ψ0[x_] = 1/25 Cos[Pi x/20] - 38/191 Cos[Pi x/25] -
         11/116 Cos[Pi x/40] - 21/23 Cos[Pi x/50] +
         79/140 Cos[3 Pi x/50] + 7/55 Cos[Pi x/100] -
         4/131 Cos[3 Pi x/100] - 95/101 Cos[Pi x/200] -
         115/166 Cos[3 Pi x/200] - 3/5 Cos[7 Pi x/200] -
         9/16 Cos[9 Pi x/200] - 100/123 Cos[11 Pi x/200] +
         12/19 Cos[13 Pi x/200];

(* system parameters *)
xM = 100; T = 80; ε = 0.001167;

(* solve the differential equation *)
nsol = NDSolve[{pde == 0, ψ[x, 0] == ψ0[x], ψ[xM, t] == ψ[-xM, t]},
               ψ[x, t], {x, -xM, xM}, {t, 0, T},
         (* set options for a solution appropriate for visualization *)
         AccuracyGoal -> 2, PrecisionGoal -> 2,
         Method -> {"MethodOfLines", "SpatialDiscretization" ->
                    {"TensorProductGrid", "DifferenceOrder" -> 10,
                     "MaxPoints" -> {1200}, "MinPoints" -> {1200}}}];
```

We visualize the solution as a density plot as well as a 3D plot. We see the “birth and death” processes for soliton-like structures [781✱] typical for this equation.

```
Show[GraphicsArray[{
(* density plot *)
DensityPlot[Evaluate[ψ[x, t] /. nsol[[1]]], {x, -xM, xM}, {t, 0, T},
           Mesh -> False, PlotRange -> All, ColorFunction -> (Hue[0.78 #]&
           DisplayFunction -> Identity, PlotPoints -> 400],
(* 3D plot *)
Plot3D[Evaluate[ψ[x, t] /. nsol[[1]]], {x, -xM, xM}, {t, 0, T},
       Mesh -> False, PlotRange -> All, PlotPoints -> 400,
       DisplayFunction -> Identity]}]]
```

The solution of the last partial differential equation was quite complicated. In general, solutions of nonlinear partial differential equations can have “any possible” shape (see [713✱] for some examples). One solution of the following coupled system of two partial differential equations (of reaction-diffusion type) has a conjectured solution exhibiting the symmetry of a Sierpinski triangle [612✱], [613✱], [614✱], [615✱], [709✱].

$$\tau \frac{\partial u(x,\, t)}{\partial t} = D_u \frac{\partial^2 u(x,\, t)}{\partial x^2} + f(u(x,\, t)) - v(x,\, t)$$

$$\frac{\partial v(x,\, t)}{\partial t} = D_v \frac{\partial^2 v(x,\, t)}{\partial x^2} + u(x,\, t).$$

Here $f(u)$ is the following nonlinear function: $f(u) = 1/2\,(\tanh((u-a)/\delta) + \tanh(a/\delta)) - u$.

The initial condition is $u(x,\, 0) = \exp(-x^2)$, $v(x,\, 0) = 0$, periodic spatial boundary conditions are imposed, and the parameter values are $a = 0.1$, $\tau = 0.34$, $\delta = 0.05$, $D_u = 1$, and $D_v = 10$ [612✱]. We use the function `NDSolve` to numerically solve the system and show a density plot of $v(x,\, t)$. (A magnified view of the solution would show a complicated fine structure [775✱], [1025✱].)

```
Module[{a = 0.1, τ = 0.34, δ = 0.05, Dv = 10, Du = 1,
        xM = 240, T = 130, pp = 700, nsol, pdeU, pdeV, u, v, x, t},
(* avoid use of high-precision arithmetic *)
Developer`SetSystemOptions["CatchMachineUnderflow" -> False];
(* nonlinear term *)
f[u_, a_, δ_] := 1/2(Tanh[(u - a)/δ] + Tanh[a/δ]) - u;
(* differential equations *)
pdeU = τ D[u[x, t], t] == Du D[u[x, t], {x, 2}] +
                          f[u[x, t], a, δ] - v[x, t];
pdeV = 1 D[v[x, t], t] == Dv D[v[x, t], {x, 2}] + u[x, t];
(* initial conditions *)
u0[x_] := Exp[-x^2];
v0[x_] := 0;
(* solve differential equations numerically *)
nsol = NDSolve[{pdeU, pdeV, u[x, 0] == u0[x], v[x, 0] == v0[x],
                u[+xM, t] == u[-xM, t], v[+xM, t] == v[-xM, t]},
               {u, v}, {x, -xM, xM}, {t, 0, T},
               (* set options appropriate for the specific problem
          and the visualization purpose *)
         MaxSteps -> 10^5, PrecisionGoal -> 2.8, AccuracyGoal -> 2.8,
               Method -> "BDF",
               Method -> {"MethodOfLines", "SpatialDiscretization" ->
                    {"TensorProductGrid", "DifferenceOrder" -> 5,
                    "MaxPoints" -> {2xM/pp}, "MinPoints" -> {2xM/pp}}}];
(* display density plot of v[x, t] *)
DensityPlot[Evaluate[v[x, t] /. nsol[[1]]], {x, -xM, xM}, {t, 0, T},
            Mesh -> False, PlotPoints -> 200, PlotRange -> All,
            ColorFunction -> (Hue[0.78 #]&)]]
```

Because of the unified underlying language of *Mathematica*, it is not only possible to perform calculations, but also to monitor the methods and algorithms used to perform the calculations. We solve the following differential equation numerically. (This differential equation is related to the s-wave phase shift of a quantum mechanical scattering problem [249✱], [258✱], [1196✱].)

$$\delta'(r) = \frac{1}{k\,r}\,(\sin(k\,r)\cos(\delta(r)) + \cos(k\,r)\sin(\delta(r)))^2$$

For small k (we use $k = 10^{-4}$) the solution $\delta(r)$ will have wide flat plateaus and short steep walls. We display the solution $\delta(r)$ as a solid line (with the corresponding ticks to the left) and the number of cumulative steps taken in the numerical solution process as a dashed line (with the corresponding ticks to the right). The correlation between the steep increases of $\delta(r)$ with the number of steps taken is obvious. (In this case, the r-values used in the solution process are easily extractable from the solution itself; in more complicated situations one can use side effects to monitor details of the algorithm and method; see Chapter 1 of the Numerics volume of the *GuideBooks* [1284✱] for more examples.)

```
(* the differential equation *)
ode[k_] = δ'[r] == (Sin[k r] Cos[δ[r]] + Cos[k r] Sin[δ[r]])^2/(k r);

(* numerical solution of the differential equation *)
nsol = NDSolve[{ode[10^-4], δ[60] == 15.70789703}, δ,
               {r, 1, 60}, MaxSteps -> 10^4, Method -> "BDF"];

Show[Graphics[{
 {GrayLevel[0], Dashing[{0.01, 0.01}],
  (* extract and number r-points *)
  Line[MapIndexed[{#1, #2[[1]]/50}&, nsol[[1, 1, 2, 3, 1]]]]},
 {GrayLevel[0], Line[Table[{r, δ[r] /. nsol[[1]]}, {r, 1, 60, 1/10}]]}}],
     (* make left and right ticks *)
     Frame -> True, Frame -> True,
     FrameTicks -> {Automatic, Automatic, False,
                    Table[{2 k, 2 k 50}, {k, 0, 8}]}]
```

The command `Table` can be used to generate a matrix. Here is a Hilbert matrix $a_{ij} = 1/(i+j+1)$ [280✱], [1139✱].

```
hilbert = Table[1/(i + j + 1), {i, 4}, {j, 4}]
```

Here, it is in the usual form.

```
hilbert // MatrixForm
```

Next, we find its eigenvalues exactly. The use of `Short` prevents a large amount of output from being printed. The structure <<*integer*>> shows the number of terms left out.

```
Eigenvalues[hilbert] // Short[#, 12]&
```

If we numerically evaluate the eigenvalues, they are, of course, much more compact.

```
Eigenvalues[N[hilbert]]
```

We get the same result if we numerically evaluate the above exact formulas for the eigenvalues.

```
N[%%]
```

Here is a slightly larger example from linear algebra. We take two random symmetric 12×12 matrices $\mathcal{H}_1$ and $\mathcal{H}_2$, form $\mathcal{H}_\alpha = (1-\alpha)\,\mathcal{H}_0 + \alpha\,\mathcal{H}_1$, and calculate the minimal distance between the eigenvalues $\lambda_k(\alpha)$ of $\mathcal{H}_\alpha$ as a function of the complex variable α. The peaks in the graphics are the branch points of the multivalued function $\lambda(\alpha)$.

```
(* two symmetric matrices *)
{ℋ0, ℋ1} = With[{n = 12}, (* generate random symmetric matrix *)
    Table[Developer`ToPackedArray[(# + Transpose[#])&[
     Table[If[i > j, 0., 2 Random[] - 1], {i, n}, {j, n}]]], {2}]];

(* minimal distance between eigenvalues *)
minEigenvalueDistance =
Compile[{{ℋ, _Complex, 2}},
        Module[{evs = Eigenvalues[ℋ], n = Length[ℋ]},
               (* distance between all pairs *)
               Min[Table[Min[Table[Abs[evs[[i]] - evs[[j]]],
                             {j, i + 1, n}]], {i, 1, n - 1}]]],
               {{Eigenvalues[_], _Complex, 1}}];
```

```
(* display eigenvalue distances over complex α-plane *)
ParametricPlot3D[{αr Cos[αφ], αr Sin[αφ], (* use logarithm *)
                  -Log[minEigenvalueDistance[(* the matrix *)
                       N[(1 - αr Exp[I αφ]) ℋ0 + αr Exp[I αφ] ℋ1]]],
                  {EdgeForm[]}}, {αr, 0, 2.5}, {αφ, 0, 2Pi},
                 PlotPoints ->  6{30, 60}, Compiled -> False,
                 BoxRatios -> {1, 1, 1/2}, PlotRange -> {-1, 5}]
```

Here is the numerical value of the Gamma function at 1/2.

```
N[Gamma[1/2]]
```

Here is the numerical value of the Bessel function $J_{3.3}(6.7)$.

```
N[BesselJ[3.3, 6.7]]
```

We can also evaluate the Bessel function for a complex argument and a complex index, for instance, $I_{3.3+0.6\,i}(6.7 - 9.5\,i)$.

```
N[BesselI[3.3 + 0.6 I, 6.7 - 9.5 I]]
```

Here is a 100-digit value of $I_{3.3+0.6\,i}(6.7 - 9.5\,i)$ (to get 100 digits, the input must have enough digits and one cannot use machine numbers as input).

```
N[BesselI[33/10 + 6/10 I, 67/10 - 95/10 I], 100]
```

The next input has 100-digit arguments. The result has fewer digits now. All digits displayed are guaranteed to be correct.

```
BesselI[N[33/10 + 6/10 I, 100], N[67/10 - 95/10 I, 100]]
```

Special functions can be evaluated for all complex arguments. This makes it possible to numerically solve the differential equation $z''(\tau) = \vartheta_2(z(\tau), q)$ where $\vartheta_2(\zeta, q)$ is an elliptic theta function. The left picture shows a solution curve of this differential equation in the complex plane and the right picture shows the corresponding recurrence plot [427✱], [242✱], [516✱], [908✱].

```
Module[{q = -0.483069 - 0.482155 I, ξ0 = 0.514593 - 0.909303 I,
        ξp0 = -0.784268 - 0.773652 I,  T = 300, pp, line, λ},
       (* solve differential equation numerically *)
       nsol = NDSolve[{z''[τ] == EllipticTheta[2, z[τ], q],
                       z[0] == ξ0, z'[0] == ξp0},
                       {z}, {τ, 0, T}, MaxSteps -> 10^5];
       (* maximal solution time *)
       T = nsol[[1, 1, 2, 1, 1, 2]];
       pp = ParametricPlot[Evaluate[{Re @ z[τ], Im @ z[τ]} /. nsol[[1]]],
                           {τ, 0, T}, PlotPoints -> 9000,
                           DisplayFunction -> Identity];
       (* extract curve *)
       line = pp[[1, 1, 1, 1]]; λ = Length[line];
       Show[GraphicsArray[
       (* solution curve in the complex plane *)
       {Graphics[MapIndexed[(* color curve *)
             {Hue[0.8 #2[[1]]/λ], Line[#1]}&, Partition[line, 2, 1]],
                 Frame -> True],
        (* recurrence plot for t ≤ 100 *)
        ContourPlot[Evaluate[Abs[z[t] - z[τ]] /. nsol[[1]]],
            {t, 0, 100}, {τ, 0, 100}, PlotPoints -> 300,
            PlotRange -> All, ContourLines -> False,
            (* use red-blue coloring scheme *)
            ColorFunction -> (RGBColor[#, 1 - #, 0]&),
            DisplayFunction -> Identity]}]]]
```

Next, we interpolate the data {{1, 2}, {2, 4}, {3, 9}, {4, 16}}.

```
Interpolation[{{1, 2}, {2, 4}, {3, 9}, {4, 16}}]
```

This input gives the value of the approximating function at $5/2$.

```
%[5/2]
```

Here is an infinite sum: $\sum_{n=1}^{\infty} n^{-2}$.

```
NSum[1/n^2, {n, 1, Infinity}]
```

Its exact value is $\pi^2 / 6$.

```
Sum[1/n^2, {n, 1, Infinity}]
```

```
N[Pi^2/6]
```

The following example is a divergent infinite sum: $\sum_{n=1}^{\infty} (n^2 - 1) / (n+1)^2$.

```
NSum[(n^2 - 1)/(n + 1)^2, {n, 1, Infinity}]
```

Here is a more difficult example using `NSum`. Euler's constant can be defined as $\gamma = \lim_{n\to\infty}(H_n - \ln(n))$, where H_n are the harmonic numbers $H_n = \sum_{k=1}^{n} 1/k$. For finite n, we have $H_n - \ln(n) = \gamma + O(1/n)$. Fortunately, Euler's constant can be expressed much more efficiently as the following limit (with error term $O(\exp(-4\,n))$ for finite n) [196★]:

$$\lim_{n\to\infty}\left(\frac{\sum_{k=0}^{\infty}\left(\frac{n^k}{k!}\right)^2 H_k}{\sum_{k=0}^{\infty}\left(\frac{n^k}{k!}\right)^2} - \log(n)\right) = \gamma$$

(The argument of the limit can be evaluated in closed form, as $K_0(2\,n)/I_0(2\,n) + \gamma$, but this is not useful for the numerical evaluation of γ). We define a function `nSum` that calls the built-in function `NSum` with options set appropriately for the two sums at hand.

```
nSum[args__] := NSum[args, (* set options appropriately *)
                    VerifyConvergence -> False, Method -> Fit,
                    PrecisionGoal -> 120, NSumTerms -> 150,
                    AccuracyGoal -> Infinity, Method -> Fit,
                    NSumExtraTerms -> 50, WorkingPrecision -> 200]
```

Now $n = 60$ yields already more than 100 correct digits for Euler's constant.

```
γ[n_] := nSum[(n^k/k!)^2 HarmonicNumber[k], {k, 0, Infinity}]/
         nSum[(n^k/k!)^2, {k, 0, Infinity}] - Log[n]
```

```
γ[60]
```

```
% - EulerGamma
```

The infinite product $\prod_{s=2}^{\infty}\left(1-s^{-2}\right)$ has an exact value of $1/2$. Here, we compute it numerically.

```
NProduct[1 - 1/s^2, {s, 2, Infinity}]
```

Mathematica can also solve the following simple minimization problem.

$$\min_{x,y}\left((x-2.56)^2+(y-3.78)^4+3.1\right)$$

```
FindMinimum[(x - 2.56)^2 + (y - 3.78)^4 + 3.1, {x, 1}, {y, 1}]
```

Here is the computation using more digits (the detailed meaning of the options `PrecisionGoal` and `AccuracyGoal` will be discussed in Chapter 1 of the Numerics volume of the *GuideBooks* [1284✱]). This time we use an objective function with exact numbers instead of machine numbers.

```
FindMinimum[(x - 256/100)^2 + (y - 378/100)^4 + 31/10, {x, 1}, {y, 1},
            PrecisionGoal -> 30, AccuracyGoal -> 30,
            WorkingPrecision -> 100] // (* shorten output *) N[#, 30]&
```

`FindRoot` solves implicit equations. Here, we solve the simple equation $\cos(x) = \sin(x)$.

```
FindRoot[Sin[x] == Cos[x], {x, 1}]
```

The result compares well with the exact root.

```
N[Pi/4]
```

Next, we look at a higher degree polynomial: $x + 2\,x^2 + 3\,x^3 + 4\,x^4 + \cdots + 66\,x^{66} = 0$.

```
poly = Sum[i x^i, {i, 66}]
```

It has 66 zeros. (By the fundamental theorem of algebra, every polynomial of nth degree has exactly n possibly complex zeros.)

```
NSolve[poly == 0] // Short[#, 10]&
```

```
Length[%]
```

Here are the Jensen disks (disks whose diameter is the segment joining complex conjugate roots) for this polynomial and all of its derivatives. (Jensen's theorem asserts that all nonreal roots of the derivative of a polynomial with real coefficients lie inside the Jensen disks of the polynomial itself [645✱], [1341✱], [1108✱], [1090✱].)

```
JensenDisks[poly_, x_] :=
 Disk[{Re[#[[1]]], 0}, Abs[Im[#[[1]]]]]& /@
      (* pairs of complex conjugate roots *)
      Partition[Cases[x /. NSolve[poly == 0, x], _Complex], 2]
```

```
Show[Graphics[MapIndexed[{GrayLevel[Mod[#2[[1]], 2]], #}&,
                (* form disks from roots *)
                JensenDisks[#, x]& /@ NestList[D[#, x]&, poly, 66]]],
                Frame -> True, AspectRatio -> Automatic]
```

Generally, a large amount of data, as in the last example, is better expressed graphically. Here, we compute all of the zeros of all polynomials of degree less than or equal to *maxDegree* with nonzero integer coefficients between −*maxInt Coeff* and *maxIntCoeff*. (For the zeros of related polynomials, see [142✱], [1017✱], [179✱], and [1016✱].)

```
allRoots[maxDegree_, maxIntCoeff_] :=
Module[{x, allMonomials, allIntegers, allCoefficientLists},
 (* the monomials *)
 allMonomials = Table[x^i, {i, 0, maxDegree}];
 (* the coefficients *)
 allIntegers = Range[-maxIntCoeff, maxIntCoeff];
 (* all possible lists of coefficients *)
 allCoefficientLists = Flatten[Outer[List,
  Sequence @@ Table[allIntegers, {maxDegree + 1}]], maxDegree];
 (* showing all roots in the complex plane *)
 Graphics[{PointSize[0.003], Point[{Re[#], Im[#]}]& /@
 (* solving all polynomials, taking roots *)
    Flatten[(Cases[NRoots[#, x], _Real | _Complex, {-1}])& /@
     DeleteCases[allMonomials.# == 0& /@ allCoefficientLists, False]]},
         PlotRange -> {{-3, 3}, {-2, 2}}, Frame -> True,
         AspectRatio -> Automatic]]
```

For `allRoots[2, 14]`, we have 24361 different polynomials and for `allRoots[5, 2]`, we have 15621 different polynomials with the following roots in the complex plane.

```
Show[GraphicsArray[{allRoots[2, 14], allRoots[5, 2]}]]
```

Now, we solve a large system of linear equations. The de Rham's function $\varphi_\alpha(x)$ fulfills the following functional equations [126✱], [691✱], [127✱], [367✱]:

$$\varphi_\alpha\left(\frac{x}{2}\right) = \alpha\,\varphi_\alpha(x)$$

$$\varphi_\alpha\left(\frac{x+1}{2}\right) = \alpha + (1-\alpha)\,\varphi_\alpha(x).$$

Discretizing the functional equations at $x = 0, \frac{1}{n}, \frac{2}{n}, \ldots, \frac{n-1}{n}, 1$ yields $2n+2$ linear equations for $2n+1$ unknowns $\varphi_\alpha(0), \varphi_\alpha\left(\frac{1}{2n}\right), \ldots, \varphi_\alpha\left(\frac{2n-1}{2n}\right), \varphi_\alpha(1)$. The function `deRhamφPoints` solves the linear equations for a given α.

```
deRhamφPoints[α_, n_] :=
Module[{φ, φs, eqs},
   (* avoid numericalization of arguments of φ *)
   SetAttributes[φ, NHoldAll];
   (* the unknowns *)
   φs = Table[φ[x], {x, 0, 1 - 1/(2n), 1/(2n)}];
   (* the linear equations *)
   eqs = N[Flatten[Table[{φ[x/2] - α φ[x] == 0,
                          φ[(x + 1)/2] - α - (1 - α) φ[x] == 0},
                        {x, 0, 1 - 1/n, 1/n}]]];
   (* 2n + 1 points of the de Rahm's function φ *)
   Apply[{First[#1], #2}&, First[Solve[eqs, φs]], {1}]]
```

The next graphic shows de Rahm's functions for various values of α. Each curve has 401 points.

```
Show[Graphics[Table[
      {Hue[0.8 α], Line[deRhamφPoints[α, 200]]}, {α, 1/20, 19/20, 1/20}]],
     Frame -> True]
```

The ability to calculate with numbers of arbitrary precision allows for straightforward investigations that otherwise would be very difficult. The following graphic shows how two orbits of the logistic map $x_{n+1} = 1 - a\,x_n^2$ move apart with increasing iteration number. We choose $x_0 = 3/7$ and $x'_0 = 3/7 + 10^{-300}$, follow 500 iterations for 200 values of a, and use 500 digits in all calculations. The resulting contour plot shows that the distance between x_n and x'_n is a sensitive function of a. (For more about the Liapunov exponent [1111✱], [123✱] of the logistic map, see [100✱], [312✱], [1286✱], [435✱], and [820✱].)

```
Module[{ε = 10^-6},
(* distance between two orbits *)
δList[a_, x0_, δx0_, n_, prec_] :=
        NestList[(1 - a #^2)&, N[x0 + δx0, prec], n] -
        NestList[(1 - a #^2)&, N[x0, prec], n];
(* data for different a *)
data = Table[Log[10, Abs[δList[a, 3/7, 10^-300, 500, 500]]],
             {a, ε, 2, (2 - ε)/200}];
(* visualize distance *)
ListContourPlot[data, MeshRange -> {{1, 500}, {0, 2}},
                Contours -> 120, ContourLines -> False,
                PlotRange -> All, ColorFunction -> (Hue[Random[]]&),
                FrameLabel -> {None, "a"}]]
```

Next, we carry out a fast Fourier transform. How does one find a built-in function that does this? The question mark `?` stands for a request for information, whereas `*` after the letters replaces any sequence of lowercase or capital letters or other characters. For example, we can find out what `Fourier` is (of course, another possibility is to look under `Fourier` in on-line version of *The Mathematica Book*, in the Help Browser).

```
?Four*
```

The following creates the values of two superimposed sine waves with different frequencies and different amplitudes. The semicolon prevents the printing of the 1000 values generated.

```
sinTable = Table[N[Sin[10 n 2Pi/1024] + 2 Sin[5 n 2Pi/1024]], {n, 1, 1024}]
```

Here is the Fourier transform (we visualize the result in the next subsection).

```
fourierTable = Fourier[sinTable];
```

We display the first 20 elements of `fourierTable`.

```
Take[fourierTable, 20] // Chop
```

Here, the two-dimensional (2D) Fourier transforms of the reciprocals of the greatest common divisor of two integers gcd(i, j) and the least common multiple lcm(i, j) for $1 \le i,\ j \le 256$ is shown.

```
Show[GraphicsArray[
Block[{$DisplayFunction = Identity},
(* display absolute value of 2D Fourier transform *)
 ListDensityPlot[Abs[Fourier[Table[1/#[i, j], {i, 256}, {j, 256}]]],
                 Mesh -> False, ColorFunction -> (Hue[0.8 #]&)]& /@
                                                      {GCD, LCM}]]]
```

Starting with *Mathematica* Version 4, in addition to the ease of programming, the fact that computations are done immediately, and the ability to plot numerical functions, *Mathematica* provides the possibility to carry out larger numerical calculations very efficiently. Larger, compiled (using the function `Compile`) numerical calculations in *Mathematica* can within a small factor achieve the speed of corresponding Fortran programs, such as those in NAG (http://www.nag.com), IMSL (http://www.imsl.com), and netlib (http://www.netlib.org) ([1093★], [1240★], [821★], and [400★]) or can even be faster. (For a rough survey of these kinds of programs, see [1137★] and [776★]).

Mathematica has a built-in (pseudo-)compiler. It generates machine-independent pseudo-code. For many numerical calculations, the use of the compiler will speed up calculations by a factor 2 to 20. Here is an example: the calculation of the Fourier spectrum of the quantum-mechanical energy spectrum of a 2D square well [1142★], [166★], [990★]. According to the Gutzwiller–Maslov theory, the Fourier spectrum contains information about the length of the classical periodic orbits [593★], [190★], [398★], [340★], [1378★], [1144★], [1157★], [1355★], [884★], [150★], [151★], and [1143★]).

This is the list of eigenvalues taken into account.

```
evList = Select[Sort[Flatten[Table[Sqrt[n^2 + m^2], {n, 60}, {m, 60}]]],
                # <= 60&];
```

The function to be calculated is $\mathrm{pl}(l) = \sum_{j=1}^{n} \exp(i\, k_j\, l)$, where the k_j are the elements of the list `evList` and n is its length. Here, for $l = 2$, the sum is calculated directly.

```
With[{l = 2.}, Abs[Total[Exp[I N[Pi evList] l]]]^2 ] // Timing
```

Compiling the function results in a considerable speed-up.

```
plCompiled = Compile[{{l, _Real}}, Evaluate[
                    Abs[Plus @@ Exp[I N[Pi evList] l]]^2]];

(* repeat calculation 10 times for a reliable timing result *)
Table[With[{l = 2.}, plCompiled[l]], {10}] // Timing
```

Here, a graphic of the absolute value of pl(l) is shown. This calculation involves nearly 3000 sums, each of them with about 3000 terms.

```
Plot[plCompiled[l], {l, 0, 20},
     PlotRange -> {0, 30000}, PlotPoints -> 80, Frame -> True,
     FrameTicks -> {Automatic, None}, Compiled -> False]
```

By using compiled functions, many numerical calculations can be carried out quite fast. Here is a modeling problem: the forest fire (model) [68★], [975★], [1177★], [889★], [1096★]. We consider a 1D array. Each element represents either a burning tree, a nonburning tree or an empty site. At each time step, a burning tree burns down creating an empty site and ignites trees that are direct neighbors and which have trees. All empty sites grow a tree with probability p. The implementation of a compiled version of a time step of the forest fire model is straightforward. In the array with the fires, trees, and empty sites, 1 stands for fire, -1 for a tree and 0 for an empty site (and ashes).

```
forestFireStepC = Compile[{{l, _Integer, 1}, p},
Module[(* use periodic boundary conditions *)
       {l1 = Append[Prepend[l, Last[l]], First[l]], l2}, l2 = l1;
       Do[(* burn trees and ignite neighbors *)
          l1[[k]] = Which[l2[[k]] == 1, 0,
                          l2[[k]] == -1 && (l2[[k - 1]] == 1 ||
                                            l2[[k + 1]] == 1), 1,
                          True, l2[[k]]], {k, 2, Length[l2] - 1}];
       (* grow new trees *)
       If[# == 0, If[Random[] < p, -1, 0], #]& /@ Take[l1, {2, -2}]]];
```

For visualizing the forest fire, we implement a function `forestFirePlot`. Fires are shown in red, trees in green, and empty sites in white.

```
forestFirePlot[data_] :=
ListDensityPlot[data, Mesh -> False, FrameTicks -> None,
        ColorFunctionScaling -> False,
        (* red for fire; green for trees; white for empty *)
        ColorFunction -> (Which[# ==  1, RGBColor[1, 0, 0],
                                # == -1, RGBColor[0, 1, 0],
                                # ==  0, RGBColor[1, 1, 1]]&)]
```

For $p = 0.22$, $p = 0.32$, and $p = 0.42$ we show the resulting fires and trees over 500 time steps for an initial array length of $L = 500$. (In average, we need $p \gtrsim 1/\ln(L)$ to keep the fire burning.) On a year-2005 computer, the calculation takes a fraction of a second for each p.

```
With[{L = 500, T = 500},
Show[GraphicsArray[
Block[{$DisplayFunction = Identity},
     (* start with same initial fires and trees *)
     Function[p, SeedRandom[111];
     forestFirePlot[NestList[forestFireStepC[#, p]&,
                Table[Random[Integer, {-1, 1}], {L}], T]]] /@
                                      {0.22, 0.32, 0.42}]]]]
```

Here is a more-complicated calculation within integer arithmetic. The sum $s_b(n)$ of the digits of an integer n in base b can be calculated in *Mathematica* in the following way.

```
digitSum[n_Integer?Positive, base_Integer /; base > 2] :=
                             Total[IntegerDigits[n, base]]
```

If one iterates $s_b(n)$ until a fixed point is reached, one gets a new function $\psi(n)$. We call it `IteratedDigitSum[n]`.

```
IteratedDigitSum[n_Integer?Positive, base_Integer /; base > 2] :=
                               FixedPoint[digitSum[#, base]&, n]
```

ψ is an arithmetic function [56✱], which means $\psi(n+m) = \psi(\psi(n)+\psi(m))$ and $\psi(n\,m) = \psi(\psi(n)\,\psi(m))$. Here, this property for two large integers is tested.

```
x = 9218359834598298562984567230456723624068502495865409134;
y = 3109579823049090378621220813796509245672098567203496722;
b = 13;

{{IteratedDigitSum[x + y, b],
  IteratedDigitSum[IteratedDigitSum[x, b] + IteratedDigitSum[y, b], b]},
 {IteratedDigitSum[x * y, b],
  IteratedDigitSum[IteratedDigitSum[x, b] * IteratedDigitSum[y, b], b]}}
```

The following pictures visualize the values of the function $\psi(n\,m)$ in the n, m-plane for base 100 and base 26.

```
Show[GraphicsArray[
ListDensityPlot[Table[IteratedDigitSum[x y, #], {x, 100}, {y, 100}],
                Mesh -> False, ColorFunction -> Hue,
                DisplayFunction -> Identity]& /@
                                (* two integer bases *) {26, 100}]]
```

As mentioned, high-precision arithmetic is a very useful tool for scientific computations. We will end this subsection with a slightly larger example. Let us deal with a simple mechanical system: a billiard ball bouncing between two types of regularly arranged circular scatterers (or a light ray reflected by perfectly mirroring circles, also called a Sinai billiard with finite horizon or a Lorentz gas [1259✱], [164✱], [1164✱]). Here are some of the scatterers shown.

```
With[{o = 3},
 Show[gr = Graphics[{Thickness[0.002],
 (* array of large and small circles *)
 Table[If[(-1)^(i + j) == 1, Circle[{i, j}, 5/8], Circle[{i, j}, 1/4]],
       {i, -o, o}, {j, -o, o}]}, AspectRatio -> Automatic]]]
```

The following functions implement the elastic scattering process of a point-shaped billiard ball between the scatterers.

```
(* nearest intersection (if any) of a ray with a circle *)
nearestIntersection[Ray[p_, d_], Circle[q_, r_]] :=
Module[{eqs = (p - q + t d).(p - q + t d) - r^2, sol},
  sol = Select[t /. Solve[eqs == 0, t], (Im[#] == 0 && # > 0)&];
  If[sol === {}, {}, p + t d /. t -> Min[sol]]]

(* reflection of a ray at the point s at a circle *)
reflect[Ray[_, d_], s_, Circle[q_, _]] :=
Module[{n = #/Sqrt[#.#]&[s - q]}, Ray[s, d - 2d.n n]]

(* the circle of next reflection for a ray *)
nextCircle[ray_, lastCircle_] :=
Module[{is}, circles =
       If[(-1)^Total[lastCircle[[1]]] === 1,
        (* big circles *)
          Join[Circle[lastCircle[[1]] + #, 5/8]& /@
                    {{2, 0}, {0, 2}, {-2, 0}, {0, -2},
                     {1, 1}, {-1, 1}, {1, -1}, {-1, -1},
                     {3, 1}, {1, 3}, {-3, 1}, {-1, 3},
                     {3, -1}, {1, -3}, {-3, -1}, {-1, -3}},
               Circle[lastCircle[[1]] + #, 1/4]& /@
                    {{1, 0}, {0, 1}, {-1, 0}, {0, -1},
                     {2, 1}, {1, 2}, {-2, 1}, {-1, 2},
                     {2, -1}, {1, -2}, {-2, -1}, {-1, -2}}],
          (* small circles *)
          Join[Circle[lastCircle[[1]] + #, 5/8]& /@
                    {{1, 0}, {0, 1}, {-1, 0}, {0, -1},
                     {2, 1}, {1, 2}, {-2, 1}, {-1, 2},
                     {2, -1}, {1, -2}, {-2, -1}, {-1, -2}},
               Circle[lastCircle[[1]] + #, 1/4]& /@
                    {{1, 1}, {-1, 1}, {1, -1}, {-1, -1}}]];
 is = DeleteCases[{nearestIntersection[ray, #], #}& /@ circles, {{}, _}];
 is[[Position[#, Min[#]]&[
          #.#& /@ ((First[#] - First[ray])& /@ is)][[1, 1]], 2]]]
```

The next function calculates *o* reflections of a billiard ball that starts at the angle *ϕ0* of the central scatterer in direction {cos(φ0), sin(φ0)}.

```
(* o reflections of a ray starting at {Cos[ϕ0], Sin[ϕ0]}
  with direction {Cos[φ0], Sin[φ0]};
  optional argument prec for high-precision *)
rayPath[ϕ0_, φ0_, o_, prec___] :=
Module[{startRay = Ray[5/8{Cos[ϕ0], Sin[ϕ0]},
                       N[{Cos[φ0], Sin[φ0]}, prec]],
        ray, lastCircle = Circle[{0, 0}, 5/8], nC, nI},
 Prepend[#, startRay]&[ray = startRay;
   (* carry out sequence of reflections *)
   Table[nC = nextCircle[ray, lastCircle];
         nI = nearestIntersection[ray, nC];
         ray = reflect[ray, nI, nC]; lastCircle = nC; ray, {o}]]]
```

The following graphic shows that the machine-precision generated ray (in blue) deviates qualitatively from the high-precision generated ray (in red) after less than 20 reflections (this is possible because of the exponential instability of the Sinai billiard [351✱], [394✱]). The high-precision calculation uses 100 digits of precision.

```
rayPathGraphic[rays_] :=
With[{λ = Length[rays]},
Graphics[{(* the circles *) gr[[1]],
          (* the reflected rays *)
          {Thickness[0.002], Table[{Hue[0.8 (k - 1)/λ],
                       Line[First /@ rays[[k]]]}, {k, λ}]}},
          PlotRange -> 3.7 {{-1, 1}, {-1, 1}}, AspectRatio -> Automatic]]

Show[{(* high-precision path *)
     rayPathGraphic[{rayPath[127/426 Pi, 121/291 Pi, 25, 100]}],
     (* machine-precision path *)
     rayPathGraphic[{rayPath[127/426 Pi, 121/291 Pi, 25]}] /.
     (* make blue path *) Hue[x_] :> Hue[x + 0.75]}]
```

The following animation shows the extreme sensitivity of the billiard path as a function of its starting direction. The trajectory starts at the rightmost point of the central circle. We color the pieces of the trajectory from red to blue.

```
Show[GraphicsArray[
Function[φ0, rayPathGraphic[{rayPath[0, φ0, 30, 120]}] /.
        Line[l_] :> (* color line segments *)
        MapIndexed[{Hue[0.78(#2[[1]] - 1)/30], Line[#]}&,
                   Partition[l, 2, 1]]] /@ {Pi/40, Pi/4, 3Pi/8}]]
```

```
Do[Show[rayPathGraphic[{rayPath[0, φ0, 30, 120]}] /.
        Line[l_] :> (* color line segments *)
        MapIndexed[{Hue[0.78(#2[[1]] - 1)/30], Line[#]}&,
                   Partition[l, 2, 1]]], {φ0, 0, Pi/2, Pi/2/90}];
```

By slightly changing the function `rayPath`, we can implement a function `fixedFinalTimeRayPath` that does not carry out a fixed number of reflections, but calculates each path for a fixed time.

```
(* carry out reflections of a ray starting at {Cos[ϕ0], Sin[ϕ0]}
   with direction {Cos[φ0], Sin[φ0]} until time T *)
fixedFinalTimeRayPath[ϕ0_, φ0_, T_, prec___] :=
Module[{ray = Ray[5/8 {Cos[ϕ0], Sin[ϕ0]},
                  N[{Cos[φ0], Sin[φ0]}, prec]], λ, Λ = 0, δL,
        nC = Circle[{0, 0}, 5/8], nI, nIO, rayBag, finalPoint},
       rayBag = {{ray, Λ}}; nIO = ray[[1]];
       (* reflect until time T has gone by *)
       While[Λ < T,  nC = nextCircle[ray, nC];
                     nI = nearestIntersection[ray, nC];
                     ray = reflect[ray, nI, nC];
                     λ = Sqrt[#.#]&[nI - nIO]; Λ = Λ + λ;
                     nIO = nI; rayBag = {rayBag, {ray, Λ}}];
       (* the flight segments *)
       rays = Partition[Flatten[rayBag], 2];
       (* cut last flight segment so that it end at time T *)
       δL = T - rays[[-2, 2]];
       finalPoint = rays[[-2, 1, 1]] + δL rays[[-2, 1, 2]];
       (* return list of flight segments *)
       Append[#[[1, 1]]& /@ Most[rays], finalPoint]]
```

We now follow 192 paths for the time 12 (we assume unit speed). The machine number calculation is about three times faster than the high-precision calculation with 120 digits.

```
Module[{ppϕ = 48, ppφ = 3, T = 12, prec = 120, d},
 (* calculate paths for machine precision
 and high precision *)
 d = Timing[Table[fixedFinalTimeRayPath[ϕ0, φ0, T, #],
                  {ϕ0, 0, 2Pi (1 - 1/ppϕ), 2Pi/ppϕ},
                  {φ0, ϕ0 - Pi/2, ϕ0 + Pi/2, Pi/ppφ}]]& /@
                               {MachinePrecision, prec};
 (* make path segment assignments and return timings *)
 {pathData, pathDataHP} = Last /@ d; First /@ d]
```

We color the paths and display them. The left graphic shows the machine arithmetic results and the right graphic shows high–precision results. The two graphic are qualitatively similar, but have different detailed paths.

```
Show[GraphicsArray[
 Graphics[{Thickness[0.002],
           MapIndexed[{Hue[#2[[1]]/8], Line[#1]}&, #, {2}]},
          PlotRange -> All, AspectRatio -> Automatic,
          Frame -> True, FrameTicks -> None]& /@
          (* machine number and high-precision data *)
                                  {pathData, pathDataHP}]]
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

■ 1.2.2 Graphics

We have already used various graphics types in the last subsection for visualizing some of the numerical results. In this subsection, we will concentrate on the graphics. We begin with a simple plot in the plane.

```
Plot[Sin[x], {x, 0, 10}]
```

The Gibbs phenomenon (see [768✱], [681✱], [1166✱], [1205✱], [850✱], [480✱], [1272✱], [554✱], [555✱], [632✱], [556✱], [548✱], [557✱], [300✱], and [1075✱]) involves the “overshoots” that occur in replacing a given function by the partial sums of its Fourier series. It is L_2-convergent, which means that the integral of the squared difference of the approximation to the given function goes to zero, but in general no pointwise convergence to the original function is achieved. If the original function $f(x)$ is of bounded variation, the series will converge pointwise to $f(x)$ at every point of continuity of $f(x)$. Here, we examine this phenomenon for the expansion of the function $\theta\left((\pi/2)^2 - x^2\right)$ ($\theta(z)$ is the Heaviside function) in terms of $\left\{(2/\pi)^{1/2}\sin(i\,x)\right\}_{i=1,2,\ldots}$. The series converge at $x = 0$ to 0 and at $x = \pi/2$ to $1/2$. We see “overshoots” near $x = 0$ and $x = \pi/2$. The left graphic shows the original step function and the first 18 partial sums over the interval $[0, \pi]$. The right graphic shows the first 120 partial sums, colored from black to white in the interval $[0, \pi/2]$ near $f(x) \approx 1$.

```
partialSum[n_, x_] :=
Sum[Sqrt[2/Pi] (1 - Cos[i Pi/2])/i Sqrt[2/Pi] Sin[i x], {i, n}]
```

```
Show[GraphicsArray[
Block[{opts = Sequence[DisplayFunction -> Identity,
                       Frame -> True, Axes -> False]},
{(* the left plot *)
 Plot[Evaluate[Table[partialSum[j, x], {j, 18}]],
      {x, 0, Pi}, Evaluate[opts], PlotRange -> All,
      PlotStyle -> {{Thickness[0.002], GrayLevel[0]}},
      Prolog -> {Thickness[0.02], GrayLevel[1/2],
                 Line[{{0, 0}, {0, 1}, {Pi/2, 1},
                       {Pi/2, 0}, {Pi, 0}}]}],
  (* the right plot *)
 Plot[Evaluate[Table[partialSum[j, x], {j, 10, 120}]],
      {x, 0, Pi/2}, Evaluate[opts], PlotPoints -> 1000,
       PlotRange -> {0.88, 1.2},
       PlotStyle -> Table[{Thickness[0.002], GrayLevel[k/120]},
                          {k, 120}]]}]]]
```

A related, not less interesting, but much less known phenomenon happens for the Fourier series description of the product of two discontinuous functions, whose product is a continuous function [844✱], [1348✱], [252✱], [1349✱].

Here are two such functions `f1` and `f2`. The left graphic shows the function `f1` in red, the function `f2` in blue, and the right product shows the product of `f1` and `f2`.

```
(* two functions with concurrent jumps *)
f1[x_] := (1 + x) UnitStep[x]  + (2 - x) UnitStep[-x]
f2[x_] := (2 + x) UnitStep[x]  + (1 - x) UnitStep[-x]
```

```
Show[GraphicsArray[
Block[{$DisplayFunction = Identity},
 {(* the two functions in red and blue *)
  Plot[{f1[x], f2[x]}, {x, -Pi, Pi}, AxesOrigin -> {0, 0},
       PlotStyle -> {RGBColor[1, 0, 0], RGBColor[0, 0, 1]}],
  (* the product of the two functions in black *)
  Plot[f1[x] f2[x], {x, -Pi, Pi}, AxesOrigin -> {0, 0}]}]]]
```

The partial sums of the two functions give Gibbs oscillations near the origin. The Fourier series coefficients of the product $c_{12}(k)$ is the convolution $\sum_{j=-o}^{o} c_1(k-j)\, c_2(j)$ of the Fourier series coefficients $c_1(k)$ and $c_2(k)$ of the two factors (we truncate the series at order o). But although the product is a continuous function, the truncated Fourier series exhibits strong oscillations (next left graphic). The right graphic shows Li's corrected version $\tilde{c}_{12}(k)$ in which the Fourier series coefficient $c_1(k-j)$ is replaced by the k,j element of the inverse of the Toeplitz matrix of the Fourier series coefficients of the inverse of `f1`.

```
(* Fourier coefficients of f1 and f2 *)
c1[k_] = If[k == 0, (3 + Pi)/2,
            ((2 - I k)(-1 + Cos[k Pi]) + k (3 + 2 Pi) Sin[k Pi])/(2 k^2 Pi)
c2[k_] = If[k == 0, (3 + Pi)/2, (Sin[(k Pi)/2] *
            (k (3 + 2 Pi) Cos[(k Pi)/2] + (-2 - I k) Sin[(k Pi)/2]))/(k^2 P
```

```
(* classical convolution for product coefficient:
  Table[Sum[If[Abs[k - j] > o, 0, c1[k - j] c2[j]], {j, -o, o}],
        {k, -o, o}] *)
(* calculated all coefficients at once *)
c12List[o_] := ListConvolve[Table[c1[j], {j, -o, o}],
                            Table[c2[j], {j, -o, o}], o + 1, 0]
(* Fourier series coefficient for 1/f1 *)
c1Inv[k_] := c1Inv[k] = N @
           If[k == 0, Log[(1 + Pi) (2 + Pi)/2]/(2 Pi), (Gamma[0, -2 I k] +
                  E^(3 I k)(Gamma[0, I k] - Gamma[0, I k (1 + Pi)]) -
                  Gamma[0, (-I) k (2 + Pi)])/(E^(2 I k)(2 Pi))];
(* concurrent jump-corrected convolution for product coefficient *)
c12SmoothedList[o_] :=
Module[{c1InvToeplitz, invMat},
       (* form Toeplitz matrix *)
       c1InvToeplitz = Table[c1Inv[n - m], {n, -o, o}, {m, -o, o}];
       invMat = Inverse[c1InvToeplitz];
       (* Fourier series coefficients *)
       invMat.Table[c2[k], {k, -o, o}]]

Show[GraphicsArray[Function[fourierSeriesList,
Plot[Evaluate[Flatten[{f1[x] f2[x], (Re @ fourierSeriesList)& /@
              {16, 32, 64, 128, 256}}]], {x, -Pi/16, Pi/16},
     PlotRange -> All, DisplayFunction -> Identity,
     PlotStyle -> (* product in black, approximations of degree
         16, 32, 64, 128, 256 from red to blue *)
     {{GrayLevel[0.8], Thickness[0.01]}, Sequence @@ ({#, Thickness[0.004]}
       {Hue[0], Hue[0.22], Hue[0.3], Hue[0.55], Hue[0.7]})}], {HoldAll}] @@
     (* classical Laurent convolution and Li's convolution *)
     {Hold[c12List[#].Table[Exp[I k x], {k, -#, #}]],
      Hold[c12SmoothedList[#].Table[Exp[I k x], {k, -#, #}]]}]]
```

The following two pictures are a visualization of the interesting limit [636✱], [485✱], [654✱].

$$f(x) = \lim_{n\to\infty} \frac{1}{n} \sum_{k=0}^{\infty} (\cos(k\,\pi\,x))^{2k} = \begin{cases} 0 & \text{if } x \text{ is irrational} \\ \frac{1}{q} & \text{if } x = \frac{p}{q} \text{ is rational, } \gcd(p, q) = 1 \end{cases}$$

The left picture represents the right-hand side (with points at all rational x with denominator less than or equal 200), and the right picture shows the convergence of the first 40 partial sums of the left-hand side.

```
With[{maxDenominator = 200, maxSeriesTerms = 40},
Show[GraphicsArray[
     (* left graphics *)
{Graphics[{PointSize[0.002],
           Union[Flatten[Table[Point[{i/j, 1/j}],
                         {j, maxDenominator}, {i, 0, j}]]]},
                PlotRange -> All, Frame -> True],
     (* generate the right plot *)
Plot[Evaluate[Table[1/n Sum[Cos[k Pi x]^(2k),
                  {k, 1, n}], {n, 1, maxSeriesTerms}]], {x, 0, 1},
       PlotPoints -> 200, PlotRange -> All,
       DisplayFunction -> Identity,
     (* different colors for n terms *)
     PlotStyle -> Table[{Thickness[0.002], Hue[0.8 i/maxSeriesTerms]},
                        {i, maxSeriesTerms}]]}]]]
```

We consider the sum $\sum_{k=0}^{n} (-1)^{s_2(l\,k)}$, where $s_2(n)$ counts the 1's in the binary representation of the integer n. This function is called `DigitCount` in *Mathematica* and exhibits fractal properties for many integers l [541✱]. The following picture shows the behavior of such sums.

```
Show[Graphics3D[{Thickness[0.002],
     Table[{Hue[l/120], Line[
      MapIndexed[{#2[[1]] - 1, l, #1}&, (* the data *)
            FoldList[Plus, 0, Table[(-1)^DigitCount[l k, 2][[1]],
                                   {k, 0, 250}]]]]}, {l, 0, 100}]}],
     BoxRatios -> {2, 1, 1}, ViewPoint -> {-0.6, -3.0, 0.8},
     Axes -> True]
```

The following plot shows a collection of half circles overlapping in a hierarchical structure. Here, we make direct use of the graphics primitive `Circle`.

```
(* upper circles *)
twoNewCircles[Circle[{x0_, y0_}, r_, {0, Pi}]] :=
    {Circle[{x0 - r, y0 - r/2}, r/2, {0, Pi}],
     Circle[{x0 + r, y0 - r/2}, r/2, {0, Pi}]};

(* lower circles *)
twoNewCircles[Circle[{x0_, y0_}, r_, {Pi, 2Pi}]] :=
   {Circle[{x0 - r, y0 + r/2}, r/2, {Pi, 2Pi}],
    Circle[{x0 + r, y0 + r/2}, r/2, {Pi, 2Pi}]};

Show[Graphics[{Thickness[0.002],
(* iterate generation *)
NestList[Flatten[twoNewCircles /@ #]&,
          {#}, 7]& /@ {Circle[{0, +1}, 1, {0, Pi}],
                       Circle[{0, -1}, 1, {Pi, 2Pi}]}}],
         AspectRatio -> 1, PlotRange -> All]
```

Here is a slightly more complicated example: the recursive filling of the area between three touching circles with circles (per Apollonius). We use an iterative rather than a direct method to calculate the new circle data via the Soddy formula [1223✱], [319✱], [128✱], [1224✱], [1255✱], [467✱], [907✱], [565✱], [566✱], [567✱], [568✱], [519✱] (for touching spheres, see [174✱], [78✱], [79✱], [1421✱]).

```
makeTouchingCircles[{p1_, p2_, p3_}, iter_, minRadius_:10^-3] :=
Module[{newton, newCircleData, rMax},
(* the derivative for the Newton method *)
newton[{{{x1_, y1_}, r1_}, {{x2_, y2_}, r2_},
        {{x3_, y3_}, r3_}}, {{xn_, yn_}, rn_}] :=
 {{#[[1]], #[[2]]}, #[[3]]}&[{xn, yn, rn} - 1/2*
 Inverse[{{(xn - x1), (yn - y1), -(rn + r1)},
          {(xn - x2), (yn - y2), -(rn + r2)},
          {(xn - x3), (yn - y3), -(rn + r3)}}].
             {(xn - x1)^2 + (yn - y1)^2 - (r1 + rn)^2,
              (xn - x2)^2 + (yn - y2)^2 - (r2 + rn)^2,
       (xn - x3)^2 + (yn - y3)^2 - (r3 + rn)^2}];
(* the next smaller circle *)
 newCircleData[{pp1_, pp2_, pp3_}] :=
{Circle @@ #, {{pp1, pp2, #[[1]]}, {pp1, pp3, #[[1]]},
  {pp2, pp3, #[[1]]}}}&[
Module[{r12, r23, r13, r1, r2, r3, startx, starty, startr,
        ε = 10^-10},
 (* radii *)
 {r12, r23, r13} = Sqrt[#.#]& /@ N[{pp1 - pp2, pp2 - pp3, pp1 - pp3}];
 r1 = ( r12 + r13 - r23)/2;
 r2 = ( r12 - r13 + r23)/2;
 r3 = (-r12 + r13 + r23)/2;
 startr = Sqrt[#.#]&[N[pp1 - ({startx, starty} =
          N[(pp1 + pp2 + pp3)/3])]] - r1;
(* iterating the Newton method *)
 FixedPoint[newton[{{pp1, r1}, {pp2, r2}, {pp3, r3}}, #]&,
            {{startx, starty}, startr},
            SameTest -> (#.#&[Flatten[#1 - #2]] < ε&)]]];
Join[Module[{r1, r2, r3}, (* the start circles *)
       {r12, r23, r13} = Sqrt[#.#]& /@ N[{p1 - p2, p2 - p3, p1 - p3}];
       r1 = ( r12 + r13 - r23)/2; r2 = (r12 - r13 + r23)/2;
       r3 = (-r12 + r13 + r23)/2; rMax = Max[r1, r2, r3];
       {Circle[p1, r1], Circle[p2, r2], Circle[p3, r3]}],
(* iterating the calculation of new circles *)
Map[First, NestList[newCircleData /@ Flatten[Map[Last,
    Select[#, (#[[1, 2]]/rMax > minRadius)&], {1}], 1]&,
        {newCircleData[{p1, p2, p3}]}, iter], {2}]]]

(* display calculated circles *)
Show[GraphicsArray[
  {#, (* color circles according to radius *) # /.
    Circle[mp_, r_?(# < 0.2&)] :> {Hue[Log[10, r]], Disk[mp, r]}}&[
   Graphics[{Thickness[0.001],
       makeTouchingCircles[{{-1, 0}, {1, 0}, {0, -1}}, 9, 0.0005]},
     PlotRange -> {(Sqrt[2] - 1) {-1, 1}, {-Sqrt[2]/2, 0}},
     AspectRatio -> Automatic]]]]
```

Here is the distribution of the logarithms of the radii of the circles [184★], [393★] from the last picture shown.

```
ListPlot[Reverse[Log[10, Sort[#1[[2]]& /@
                 (* extract all circles from the last picture *)
                 Cases[%[[1]], _Circle, Infinity]]]], PlotRange -> All]
```

Discrete data can also be displayed. For example, here is again a plot of a Fourier transform of a superposition of sin functions. We clearly see the frequency and amplitude ratios according to the signal.

```
fourierTable =
 Fourier[Table[(* the signal *)
               Sum[Sin[16. k n 2Pi/1024]/k, {k, 20}], {n, 1, 1024}]];
```

```
ListPlot[(* add x-values *)
         Flatten[MapIndexed[{{#2[[1]], 0}, {#2[[1]], #1}, {#2[[1]], 0}}&,
                            Abs[Take[fourierTable, 512]]], 1],
         PlotRange -> All, PlotJoined -> True]
```

The next picture shows the first 30 partial sums of the generalized Weierstrass function $\sum_{k=1}^{n} k^{-2} \exp(i\, k^3\, z)$ in the complex plane. In the limit $n \to \infty$, the resulting curve is nowhere differentiable [261★].

```
Module[{l = 10000, cf, lines},
(* fast calculation of the cumulative sums *)
cf = Compile[{{z, _Complex}},
             Rest[FoldList[Plus, 0, Table[Exp[I k^3 z]/k^2, {k, 30}]]]];
(* the lines *)
lines = Line /@ Transpose[Table[{Re[#], Im[#]}& /@ cf[ϑ],
                                {ϑ, 0., 2.Pi, 2.Pi/l}]];
(* the graphics *)
Show[Graphics[{Reverse[
MapIndexed[{Hue[3 #2[[1]]/40], (* smoother lines are thicker *)
            Thickness[0.002 #2[[1]]], #1}&, Reverse[lines]]]}],
      PlotRange -> All, Frame -> True, FrameTicks -> None,
      AspectRatio -> Automatic, Background -> GrayLevel[0.8]]]
```

Here is a typical three-dimensional (3D) plot. By default, the surface is illuminated with three colored light sources.

```
Plot3D[Sin[x^2 + y^2]/(x^2 + y^2), {x, -4, 4}, {y, -4, 4},
       PlotPoints -> 35, PlotRange -> All]
```

The coloring and many other details can be varied as desired.

```
Plot3D[{Sin[x^2 + y^2]/(x^2 + y^2), Hue[Sqrt[x^2 + y^2]/Sqrt[32]]},
       {x, -4, 4}, {y, -4, 4}, PlotPoints -> 40, PlotRange -> All,
       Axes -> None, Boxed -> False, Mesh -> False]
```

We now plot a sphere given in a parametric form analogous to the one for the circle above.

```
ParametricPlot3D[{Cos[φ] Sin[ϑ], Sin[φ] Sin[ϑ], Cos[ϑ]},
                 {φ, 0, 2 Pi}, {ϑ, 0, Pi}]
```

Here is a more complicated surface. It is based on the parametrization of a torus. This time we do not show the edges of the polygons.

```
torus[φ_, ϑ_, R_, r_, color_] :=
{R Cos[φ] + r Cos[φ] Cos[ϑ], R Sin[φ] + r Sin[φ] Cos[ϑ], r Sin[ϑ], color}

ParametricPlot3D[Evaluate[(* modify torus parametrization *)
   torus[φ + Sin[7 φ]/3, 4 φ + ϑ, 3 + Sin[φ]/3 + Sin[ϑ]/5, 1 + Sin[11 φ]/3,
         (* surface coloring *)
         {EdgeForm[], SurfaceColor[RGBColor[0.9, 0, 0.4],
                                   RGBColor[0.3, 0.4, 0], 2.3]}]],
        {φ, 0, 2 Pi}, {ϑ, 0, Pi},
    (* set options *) PlotPoints -> {300, 40}, Boxed -> False,
    Axes -> False, ViewPoint -> {0, 0, 0.51}]
```

Next, we visualize a complicated closed surface with infinitely many holes. It is implicitly defined by

$$\cos\left(\frac{x+y}{x^2+y^2+z^2}\right)+\cos\left(\frac{x+z}{x^2+y^2+z^2}\right)+\cos\left(\frac{y+z}{x^2+y^2+z^2}\right)+$$
$$\sin\left(\frac{x-y}{x^2+y^2+z^2}\right)-\sin\left(\frac{x-z}{x^2+y^2+z^2}\right)+\sin\left(\frac{y-z}{x^2+y^2+z^2}\right)=0.$$

Because the denominators of the arguments of the trigonometric functions vanish faster than the numerators when approaching the origin, the surface becomes quite complicated near the origin. The following code generates an approximation of this surface. We use the function `ContourPlot3D` from the package ``Graphics`ContourPlot3D` ``.

```
Needs["Graphics`ContourPlot3D`"]

Module[{n = 1, pp0 = 32, ppR = 22, cp, polys},
(* define 3D contour plot of function with
   {x, y, z} --> {x, y, z}/(x^2 + y^2 + z^2) *)
cp[pp_] := cp[pp] = Cases[
ContourPlot3D[Cos[x + y] + Cos[x + z] + Cos[y + z] +
              Sin[x - y] - Sin[x - z] + Sin[y - z],
              {x, -Pi, Pi}, {y, -Pi, Pi}, {z, -Pi, Pi},
              MaxRecursion -> 0, PlotPoints -> pp, Contours -> {0},
              DisplayFunction -> Identity], _Polygon, Infinity];
(* the polygons *)
polys = Table[Map[# + 2.Pi{i, j, k}&,
                  If[i == j == k == 0, cp[pp0], cp[ppR]], {-2}],
               {i, -n, n}, {j, -n, n}, {k, -n, n}] // Flatten;
(* display polygons *)
Show[Graphics3D[{EdgeForm[], SurfaceColor[Hue[0.22], Hue[0.02], 2.6],
                 polys} /. (* invert *)
                         Polygon[l_] :> Polygon[#/#.#& /@ l]],
     PlotRange -> All, Boxed -> False]]
```

Cutting the surface along the *x*,*y*-plane and removing the upper part shows its complicated structure near the origin.

```
Show[%, PlotRange -> {All, All, {-3/4, 0}}, ViewPoint -> {0, 0, 3}]
```

The results of such functions as `Plot`, `Plot3D`, and `ParametricPlot3D` are composed of graphics primitives that can be further manipulated by *Mathematica*. In the following plot, we use `ParametricPlot3D` to subdivide the sides of a cube (leftmost image). These side faces are then pulled toward the center of the cube by an amount corresponding to their distance to the center. The upper right image shows the resulting surface reflected in a sphere. To see inside these surfaces, we have made holes in the polygons.

```
Show[GraphicsArray[Map[
Function[α, Graphics3D[{EdgeForm[Thickness[0.001]],
SurfaceColor[Hue[0.12], Hue[0], 2.2],
(* fit cube in unit cube *)
Function[polys, Module[{rMax = Max[
         Sqrt[#.#]& /@ Level[Cases[polys, _Polygon, Infinity], {-2}]]},
         Map[#/rMax&, polys, {-1}]]] @
(* making holed polygons on deformed surfaces *)
(Function[x, Module[{mp = Mean[x[[1]]]},
 {Polygon[(mp + 0.2 (# - mp))& /@ x[[1]]],
 MapThread[Polygon[Join[#1, Reverse[#2]]]&,
 {Partition[Append[#, First[#]]&[
            (mp + 0.8 (# - mp))& /@ x[[1]]], 2, 1],
  Partition[Append[#, First[#]]&[
            (mp + 0.5 (# - mp))& /@ x[[1]]], 2, 1]}]}]] /@
            Map[(* this deforms the faces *)
                (#/Sqrt[#.#] Sqrt[#.#]^α)&, Join @@
(* making a cube; every side has 6×6 polygons *)
Apply[ParametricPlot3D[##,
      PlotPoints -> 7, DisplayFunction -> Identity][[1]]&,
 {#[[1]], Flatten[Append[{#[[2, 1]]}, {-1, 1}]],
          Flatten[Append[{#[[2, 2]]}, {-1, 1}]]}& /@
       ({#, Cases[#, _Symbol]}& /@
        Select[Flatten[Outer[List, {x, 1, -1}, {y, 1, -1}, {z, 1, -1}], 2],
        Length[Cases[#, _Symbol]] == 2&]), {1}], {-2}])},
       Axes -> False, PlotRange -> {{-1, 1}, {-1, 1}, {-1, 1}}]],
     (* the values for the pure function parameter α *){-3, -1, 1, 2}],
         GraphicsSpacing -> -0.05]]
```

Images can also be created directly from graphics primitives—such as points, lines, and polygons—rather than as plots of functions. Here is a problem that was investigated already by Kepler. It involves the recursive subdivision of a regular pentagon according to the following visualized rule [385✱], [870✱]. (Here, we also implement the routines needed for the next two images.) The implementation itself is straightforward. For clarity, we do not enclose all pieces of the code in scoping constructs but rather use global variables like `fac` and *fac*. We discuss similar graphics in detail in Chapter 1 of the Graphics volume of the *GuideBooks* [1283✱].

```
startPentagon = Polygon[Table[{Cos[x], Sin[x]},
                              {x, Pi/2, -11/10 Pi, -2Pi/5}]];

(* makes a vector, perpendicular to vec *)
perpendicular[vec_] := #/Sqrt[#.#]&[{vec[[2]], -vec[[1]]}] // N;

(* for the pentagon-specific constants *)
{fac, fac} = N[{1/(2 + 2 Sin[18 Degree]), Sin[72 Degree]}];

(* new points for making smaller pentagon *)
threeNewPoints[{p1_, p2_}] :=
{p1 + fac (p2 - p1), p2 + fac (p1 - p2),
 (p1 + p2)/2 + fac fac Sqrt[(p2 - p1).(p2 - p1)] perpendicular[p2 - p1]};

sixNewPentagons[Polygon[l_]] :=
(* treating every side *)
Module[{p1, p2, p3, p4, p5, p6, p7, p8, p9, p10,
        p11, p12, p13, p14, p15, p16, p17, p18, p19, p20},
       (* the new points *)
       {p1, p2, p3, p4, p5} = l;
       {{p6, p7, p16}, {p8, p9, p17}, {p10, p11, p18},
        {p12, p13, p19}, {p14, p15, p20}} =
       threeNewPoints /@ Partition[Append[l, First[l]], 2, 1];
       (* the six new pentagons *) Polygon /@
       {{p1, p6, p16, p20, p15}, {p7, p2, p8, p17, p16},
        {p9, p3, p10, p18, p17}, {p11, p4, p12, p19, p18},
        {p13, p5, p14, p20, p19}, {p16, p17, p18, p19, p20}}]

Show[GraphicsArray[
{Graphics[startPentagon, AspectRatio -> Automatic],
 Graphics[sixNewPentagons[startPentagon],
          AspectRatio -> Automatic]}] /.
                    Polygon[l_] :> Line[Append[l, First[l]]]]
```

If we repeat this subdivision four times, we get a figure consisting of $6^4 = 1296$ pentagons in interesting positions.

```
subdividedPentagons[0] =  {startPentagon};
subdividedPentagons[k_] := subdividedPentagons[k] =
          Flatten[sixNewPentagons /@ subdividedPentagons[k - 1]]

Show[Graphics[{Thickness[0.001],
               Line[Append[#, First[#]]]& @@@ subdividedPentagons[4]} // N]
     AspectRatio -> Automatic]
```

Now, we color the pentagons in each step with some color and stack them up.

```
Show[Graphics[Table[{Hue[k/5], subdividedPentagons[k]} // N,
                    {k, 0, 4}]], AspectRatio -> Automatic]
```

Here, we project Kepler's recursive subdivision of a pentagon onto a sphere.

```
toSphere[{x_, y_}] := Function[{φ, ϑ},
       {Cos[φ] Sin[ϑ], Sin[φ] Sin[ϑ], Cos[ϑ]}][
                         ArcTan[x, y], Sqrt[x^2 + y^2] N[Pi]]

(* a function that cuts a hole in a polygon *)
makeHole[Polygon[l_], factor_] :=
Module[{mp = Mean[l], newl, nOld, nNew},
 (* inner points *) newl = (mp + factor(# - mp))& /@ l;
{nOld, nNew} = Partition[Append[#, First[#]], 2, 1]& /@ {l, newl};
{MapThread[Polygon[Join[#1, Reverse[#2]]]&, {nOld, nNew}]}]
```

```
Show[Graphics3D[{EdgeForm[], Thickness[0.001],
     {SurfaceColor[Hue[Random[]], Hue[Random[]], 3 Random[]],
     makeHole[#, 0.8]}& /@ Map[toSphere, subdividedPentagons[4], {3}]}],
     Boxed -> False]
```

Several 3D figures can be directly constructed from polygons. Here is a fractal sign post.

```
(* normalize a vector *)
normalize[a_List] = a/Sqrt[a.a];

(* make one elementary part of the sign post *)
post[α_, dir_, ortho_, size_] :=
Module[{dir1, orthoh, ortho1, bi1, p1, p2, p3, p4, p5, p6, p7, p8, p9,
        s1 = 1, s2 = 0.3, s3 = 0.2, s4 = 1.2, h1, h2, h3, h4, h5},
       (* direction the new sign will point to *)
       dir1 = normalize[dir];
       (* first orthogonal direction *)
       ortho1 = normalize[normalize[ortho] +
                 normalize[Cross[dir, ortho]]];
       (* second orthogonal direction *)
       bi = normalize[Cross[dir1, ortho1]];
       h1 = s2 size ortho1; h2 = s2 size bi;
       h3 = s3 size ortho1; h4 = s3 size bi;
       h5 = s1 size dir1;
       p1 = α + h1; p2 = α + h2; p3 = α - h1; p4 = α - h2;
       p5 = α + h3 + h5; p6 = α + h4 + h5; p7 = α - h3 + h5;
       p8 = α - h4 + h5; p9 = α + s4 size dir1;
       (* polygons forming the next generation *)
       Polygon /@ {{p1, p4, p8, p5}, {p4, p3, p7, p8}, {p3, p2, p6, p7},
                   {p2, p1, p5, p6}, {p5, p9, p8}, {p8, p7, p9},
                   {p6, p7, p9}, {p5, p6, p9}}]

       (* the start part *)
       postHierarchy[0] = {post[{0., 0., 0.}, {0., 0., 1.}, {1., 0., 0.}, 1
       (* add new parts at the sides *)
       postHierarchy[i_] := postHierarchy[i] =
           (post @@ newData[#, 0.4^i])& /@ Flatten[(Take[#, 4]& /@
                                               postHierarchy[i - 1])];
       (* iterate the process *)
newData[poly_Polygon, size_] :=
       Module[{f = poly[[1]], ortho, dir},
              ortho = (f[[1]] + f[[2]])/2 - (f[[3]] + f[[4]])/2;
              p = (f[[3]] + f[[4]])/2 + 0.2 ortho;
              dir = -Cross[f[[1]] - f[[2]], f[[1]] - f[[4]]];
              {p, dir, ortho, size}]

Show[Graphics3D[{EdgeForm[Thickness[0.001]],
      SurfaceColor[Hue[0.11], Hue[0.10], 2],
      Table[postHierarchy[i], {i, 0, 4}]}],
     AspectRatio -> Automatic, Boxed -> False, PlotRange -> All]
```

With *Mathematica*'s symbolic, numerical, and graphical capabilities, much more complicated images with many more points and polygons can be created and displayed. However, this often requires some more CPU time and memory resources. Here is an example of such an image involving a flower made out of a dodecahedron. It consists of 6300 polygons.

```
Needs["Graphics`Polyhedra`"];

Module[{preCup, preBlossom, cup, blossom, allPolys, rotation,
        mat, rotMat, vec = {0.324919, 0.324919, 0.180513}},
(* the elementary parts, made with ParametricPlot3D *)
preCup =
ParametricPlot3D[{Sin[ϑ]^2/3 Cos[φ], Sin[ϑ]^2/3 Sin[φ], ϑ},
                 {φ, -Pi, Pi}, {ϑ, 0, Pi/2}, PlotPoints -> {26, 8},
                 DisplayFunction -> Identity];
preBlossom =
ParametricPlot3D[{(2 - 5/3 Sin[ϑ]) Cos[(Pi - ϑ)/(Pi/2) φ],
                  (2 - 5/3 Sin[ϑ]) Sin[(Pi - ϑ)/(Pi/2) φ],
                  Pi/2 + 2(ϑ - Pi/2)},
                 {φ, -Pi/5, Pi/5}, {ϑ, Pi/2, Pi},
                 PlotPoints -> {6, 15}, DisplayFunction -> Identity];
(* a rotation matrix *)
mat = {{Cos[#], Sin[#], 0}, {-Sin[#], Cos[#], 0}, {0, 0, 1}}&[Pi/5.];
(* the cup *)
cup = Map[vec (mat.#)&, preCup[[1]], {-2}];
(* one part of the blossom *)
blossom[0] = Map[vec (mat.#)&, preBlossom[[1]], {-2}];
(* rotation matrices for other five subparts of one part *)
Do[ℛ[i] = {{ Cos[2Pi/5 i], Sin[2Pi/5 i], 0},
           {-Sin[2Pi/5 i], Cos[2Pi/5 i], 0}, {0, 0, 1}} // N, {i, 4}];
(* the blossom *)
Do[blossom[i] = Map[ℛ[i].#&, blossom[0], {-2}], {i, 4}];
allPolys = Flatten[{cup, Table[blossom[i], {i, 0, 4}]}];
(* rotation matrices for other eleven parts *)
With[{aMat = Table[𝒶[k, l][i], {k, 3}, {l, 3}]},
(* rotation matrices for other faces of dodecahedron *)
Do[rotation[i] = (aMat /. Solve[Flatten[Table[Thread[
    aMat.Polyhedron[Dodecahedron][[1, 1, 1, j]] ==
        Polyhedron[Dodecahedron][[1, i, 1, j]]], {j, 3}]],
              Flatten[aMat]])[[1]], {i, 12}]];
(* display cup and blossoms *)
Show[Graphics3D[{EdgeForm[{Hue[0], Thickness[0.001]}],
                 SurfaceColor[RGBColor[0, 0.8, 0.2],
                              RGBColor[0.1, 0.9, 0.4], 1],
                 Table[Map[rotation[i].#&, allPolys, {-2}], {i, 12}]}],
     Boxed -> False, PlotRange -> All, ViewPoint -> {2.1, -2.4, 2.3}]]
```

Mathematica has built-in functions for many kinds of graphics. The following picture shows a contour plot of the absolute value of the Gauss map $z \longrightarrow 1/z - \lfloor 1/z \rfloor$ over the complex z-plane.

```
ContourPlot[Abs[1/(x + I y) - Floor[1/(x + I y)]],
            {x, -1.1, 1.1}, {y, -1.1, 1.1},
            PlotPoints -> 400, ColorFunction -> Hue,
            ContourStyle -> {Thickness[0.001]}]
```

In the following, we use a sum of three Gauss maps to create an animation. Let $\{z\}$ denote the fractional part of z. We will animate a contour plot of the function

$$f(z) = \left|\left\{\frac{1}{(z-1)^{\alpha}}\right\}\right| + \left|\left\{\frac{1}{\left(e^{2i\pi/3}\,z-1\right)^{\alpha}}\right\}\right| + \left|\left\{\frac{1}{\left(e^{4i\pi/3}\,z-1\right)^{\alpha}}\right\}\right|$$

as the parameter α varies from $1/2$ to 3.

```
fractionalPartContourPlot[α_, opts___] :=
Module[{r = 2.15, ring, color, cp},
(* cut out circular area *)
ring = {GrayLevel[1],
        Polygon[Join[Table[3.05 {Cos[φ], Sin[φ]}, {φ, 0, 2Pi, 2Pi/200}],
        Reverse[Table[r {Cos[φ], Sin[φ]}, {φ, 0, 2Pi, 2Pi/200}]]]]};
(* coloring for the contour lines *)
color[l_] := {Hue[α + 0.8 Sqrt[#.#&[Total[l]/2]]], Line[l]};
(* make the contour plot *)
cp = ContourPlot[Evaluate[Sum[
    Abs[FractionalPart[(Exp[I φ](zr + I zi) - 1)^-α]],
                                {φ, 0, -4/3Pi, -2/3Pi}]],
          {zi, -r, r}, {zr, -r, r}, PlotPoints -> 301,
PlotRange -> All, DisplayFunction -> Identity, Frame -> False,
Epilog -> ring, ColorFunctionScaling -> False,
Contours -> Table[ξ, {ξ, 0, 9/2, 3/10}],
(* color contour zones alternatingly *)
ColorFunction :> (Which[# == 0, RGBColor[1, 0, 0],
                        # == 1, RGBColor[0, 0, 1]]&[
                               Mod[Ceiling[# 10], 2]]&)];
(* display the contour plot with re-colored contour lines *)
Show[Graphics[cp] /. Line[l_] :> (color /@ Partition[l, 2, 1]),
     opts, DisplayFunction -> $DisplayFunction,
     PlotRange -> {{-r, r}, {-r, r}}]]

(* show 3×3-array of graphics for various α *)
Show[GraphicsArray[fractionalPartContourPlot[#,
                         DisplayFunction -> Identity]& /@ #,
                        GraphicsSpacing -> 0.2]]& /@
                           Partition[Table[α, {α, 1/2, 3, 5/2/8}], 3]
```

```
(* generate frames of the animation *)
Do[fractionalPartContourPlot[α], {α, 1/2, 3, 5/2/75}];
```

Let us give a few more graphics examples. Here is an iterative construction of a fractal tree using *n* iteration levels.

```
FractalTree[n_] :=
With[{α = 0.65, β = 0.87, γ = 0.46, δ = 0.8},
     Graphics[Polygon[Join[{{1.35, -0.2}, {1.1, 0}},
     Map[{0.5, 0} + (* deform the pattern *)
          1/(1 - 0.4 Cos[2 ArcTan @@ (# - {0.5, 0})]^2)*
         (# - {0.5, 0})&, Flatten[MapIndexed[
            If[#2[[1]] == 1 || #2[[1]] == 5^n, #1, Drop[#1, -1]]&,
      {#[[1]], #[[1]] + γ(#[[4]] - #[[1]]) + δ(#[[2]] - #[[1]]),
       #[[4]]} & /@ (Function[p, Module[{mp}, mp = Mean[p];
                                   (mp + β(# - mp))& /@ p]] /@
      Nest[Flatten[(* just a "random" fancy form;
           many others are possible here *)
      Apply[{{#1, #5, #11, #6}, {#6, #2, #7, #12},
             {#12, #11, #10, #9}, {#9, #7, #3, #8},
             {#8, #10, #5, #4}}& @@
        {#1, #2, #3, #4, #4 + α(#1 - #4), #1 + α(#2 - #1),
         #2 + α(#3 - #2), #3 + α(#4 - #3),
         #4 + (1 - α)(#1 + #3 - 2#4),
         (2α - 1)#1 + (1 - α)(#2 + #4), α(#1 + #3 - 2#4) + #4,
         (1 - α)#1 + α #3}&, #, {1}], 1]&,
           {{{1, 0}, {1, 1}, {0, 1}, {0, 0}}}, n]), {1}], 1]],
                                 {{-0.1, 0}, {-0.35, -0.2}}]],
         AspectRatio -> Automatic]]
```

These are the first three levels of growth.

```
Show[GraphicsArray[Table[FractalTree[k], {k, 4}],
                   GraphicsSpacing -> -0.05]]
```

The growth of this tree proceeds deterministically. We show the fifth level separately because of the fine details involved.

```
Show[FractalTree[5]]
```

The next graphic is a fractal based on the iteration of the function $z \to 4\,(1+i)\left((3+i+5\,(1+2\,i)\,z/c)^{-1-i}\right)^{2\,i}$. We display the number of iterations carried out until the condition $|z| > 100$ is fulfilled as a function of the complex parameter c.

```
DensityPlot[Function[c, (* iterate until |z| > 100 *)
             Module[{k = 1, z = 1.0 + 1.0 I, max = 100., maxk = 100},
                While[k < maxk && Abs[z] < max, k++;
                      z = (1/4 + I/4)((3 + I + (1/5 + 2I/5)*
                           z/c)^(-1 - I))^(2I)]; k]][cx + I cy],
            {cx, -2, 2.25}, {cy, -3.4, 1.5},
            ColorFunction -> (Hue[Pi #]&), Mesh -> False,
            ColorFunctionScaling -> False, FrameTicks -> None,
            (* use many points *) PlotPoints -> 600, Compiled -> True]
```

We now iterate a random subdivision of two triangles. The thickness of the edges of the triangles decreases with each iteration.

```
With[{level = 10},
Show[Graphics[Reverse[
 MapIndexed[{Hue[#2[[1]]/9], Thickness[0.03/#2[[1]]],
             Line[Append[#, First[#]]]& /@ #1}&,
             NestList[Flatten[((* iteration of the subdivision *)
 Apply[Function[{d1, d2, d3},
 (* divide longest side *)
 Which[# == 1, {{d1, #, d3}, {d2, #, d3}}&[
              d1 + Random[Real, {0.25, 0.75}] (d2 - d1)],
       # == 2, {{d1, #, d2}, {d3, #, d2}}&[
              d1 + Random[Real, {0.25, 0.75}] (d3 - d1)],
       # == 3, {{d2, #, d1}, {d3, #, d1}}&[
              d2 + Random[Real, {0.25, 0.75}] (d3 - d2)]]&[
 (* position of longest side *)
 (Position[#, Max[#]]&[#.#& /@
 {#1 - #2, #1 - #3, #2 - #3}&[d1, d2, d3]])[[1, 1]]]],
  #, {1}]), 1]&, (* start triangles *)
             {{{0, -1}, {0, 1}, {3, -1}},
              {{0, +1}, {3, 1}, {3, -1}}} // N, level], {1}]]],
      AspectRatio -> Automatic, PlotRange -> All]]
```

Lines can also be drawn in 3D space, as in this abstract branch.

```
Module[{extend},
(* add some new hairs *)
extend[x_, ω_] :=
Module[{c = N[Cos[ω]], s = N[Sin[ω]], vOld, vm, vPerp, v, α, β},
 (* orthogonal directions *)
 {vOld, vm} = {x[[2]] - x[[1, 1]], x[[1, 2]] - x[[1, 1]]};
 vPerp = #/Sqrt[#.#]&[vOld - vm vm.vOld];
 v3 = #/Sqrt[#.#]&[Cross[vm, vPerp]]; {α, β} = x[[1]];
 (* the new hairs *)
Function[f, {{β, β + #/Sqrt[#.#]&[c vm + s vPerp + f s v3]},
               α}] /@ {0, 1, -1}];

(* display iterated addition of hairs *)
Show[Graphics3D[Rest[
MapIndexed[{Hue[(#2[[1]] - 2)/8],
                (* color and add various thickness *)
                Thickness[2^(-#2[[1]] - 3)], Line /@ #1}&,
Map[First, (* iterate the process *)
FoldList[Flatten[Function[x, extend[x, #2]] /@ #1, 1]&,
          {{{{0, 0, 0}, {0, 0, 1}},
             {-Sin[28. Degree], 0, Cos[28. Degree]}}} // N,
         {30, 25, 20, 16, 11, 8, 5} Degree], {-3}]]]],
          PlotRange -> All, Boxed -> False]]
```

In the following construction, the edges of Platonic solids are taken and rotated continuously outward until they have the position of an edge again. (See [438✱] for a description of the resulting surfaces.)

```
Needs["Graphics`Polyhedra`"]
```

```
RotatedSideWireFrame[platonicSolid:
       (Cube | Tetrahedron | Octahedron | Dodecahedron | Icosahedron),
       steps_Integer?(# > 2&), opts___] :=
Module[{l = Length[Faces[platonicSolid][[1]]] - 1, makeLines,
        combis, allLines, s = steps},
(* rotate edges outwards *)
makeLines[points_] :=
 Module[{l = Length[points]},
      Join @@ Table[{(1 + t) points[[1]],
         (1 - (l - 2) (t - (i - 2)/(l - 2))) (1 + t) points[[i]] +
         (l - 2) (t - (i - 2)/(l - 2)) (1 + t) points[[i + 1]]},
        {i, 2, l - 1}, {t, (i - 2)/(l - 2), (i - 1)/(l - 2), 1/(l - 2)/s}]]
(* all possible combinations of points to rotate about *)
combis = Join[Flatten[Table[RotateRight[#, i], {i, 0, l}]& /@
             Faces[platonicSolid], 1],
(* rotate in both directions *)
Flatten[Table[RotateRight[#, i], {i, 0, l}]& /@
        (Reverse /@ Faces[platonicSolid]), 1]];
(* all lines *)
allLines = makeLines /@ Map[#/Sqrt[#.#]&[N[Vertices[platonicSolid][[#]]]]&,
                              combis, {2}];
(* display all rotated lines *)
Show[Graphics3D[{Thickness[0.001],
   MapIndexed[{Hue[(#2[[2]] - 1)/s 3/4], Line[#1]}&, allLines, {2}],
   MapIndexed[{Hue[(#2[[2]] - 1)/s 3/4], Line[#1]}&,
              Transpose[allLines, {1, 3, 2, 4}], {2}]}], opts,
      PlotRange -> {{-2, 2}, {-2, 2}, {-2, 2}},
      Boxed -> False, ViewPoint -> {2, 2, 2}]]

Show[GraphicsArray[ (* all five Platonic solids *)
Apply[RotatedSideWireFrame[##, DisplayFunction -> Identity]&,
      {{Tetrahedron, 16}, {Octahedron, 15}, {Cube, 12},
       {Dodecahedron, 8}, {Icosahedron, 10}}, {1}],
      GraphicsSpacing -> -0.25]]
```

Our next example involves an iterated construction using equilateral triangles. Each new magnified or shrunken triangle is attached to an old vertex. It is drawn in the plane formed by the normal to the old triangle and the line connecting the center of the old triangle to the vertex.

```
(* the new triangles at the correct position *)
newTriangle[x_, fac_] :=
Module[{mpo = Mean[x], mp2, mp3, dir1, dir2, poly2, poly3},
       (* midpoint *)
       mp2 = x[[2]] + fac (x[[2]] - mpo);
       (* orthogonal directions *)
       dir1 = mpo - x[[2]];
       dir2 = Cross[x[[2]] - x[[1]], x[[2]] - x[[3]]];
       dir2 = #/Sqrt[#.#]&[dir2];
       poly2 = Table[mp2 + fac (Cos[φ] dir1 + Sin[φ] dir2),
                     {φ, 0, 2 2Pi/3, 2Pi/3}] // N;
       mp3 = x[[3]] + fac (x[[3]] - mpo);
       dir1 = mpo - x[[3]];
       poly3 = Table[mp3 + fac Cos[φ] dir1 + fac Sin[φ] dir2,
                     {φ, 0, 2 2Pi/3, 2Pi/3}] // N; {poly2, poly3}];
(* make three new polygons *)
three[x_] := N[
{x, Map[({{-1, +Sqrt[3], 0}, {-Sqrt[3], -1, 0}, {0, 0, 1}}/2).#&, x, {-2}],
    Map[({{-1, -Sqrt[3], 0}, {+Sqrt[3], -1, 0}, {0, 0, 1}}/2).#&, x, {-2}]}
```

Here is a visualization of the first two steps of attaching new triangles.

```
Show[Graphics3D[
Join[{{Hue[0], {Polygon[{{2, 0, 0}, {-1, Sqrt[3], 0},
                         {-1, -Sqrt[3], 0}}/2.]}}},
MapIndexed[{Hue[#2[[1]]/10], (* add color *)
            Polygon /@ Flatten[#1, 1]}&,
  Transpose[three[FoldList[ (* iterate the construction *)
      Flatten[Function[x, newTriangle[x, #2]] /@ #1, 1]&,
            {{{-2, 2Sqrt[3], 0}, {-5, 5Sqrt[3], -2Sqrt[3]},
              {-5, 5Sqrt[3], 2Sqrt[3]}}/4 // N}, {1}]]]]]]],
  PlotRange -> All, Lighting -> False,
  Boxed -> True, ViewPoint -> {3, 3, 3}]
```

Now, this process is repeated eight times.

```
Show[Graphics3D[
Join[{{Hue[0], {Polygon[{{1, 0, 0}, {-1/2, 3^(1/2)/2, 0},
                         {-1/2, -3^(1/2)/2, 0}}]}}},
MapIndexed[{Hue[#2[[1]]/4], (* color the triangles *)
            Polygon /@ Flatten[#1, 1]}&,
  Transpose[three[FoldList[ (* iterate the construction *)
      Flatten[Function[x, newTriangle[x, #2]] /@ #1, 1]&,
            {{{-1/2, 1/2 Sqrt[3], 0          },
              {-5/4, 5/4 Sqrt[3], -1/2 Sqrt[3]},
              {-5/4, 5/4 Sqrt[3], +1/2 Sqrt[3]}} // N},
 {1, 1, 1, 1, 1, 1, 1}]]]]]]],
  PlotRange -> All, Lighting -> False, Boxed -> False,
  ViewPoint -> {3, 3, 3}]
```

Mathematica also includes functions to manipulate a graphic as a whole without explicitly manipulating, removing, or adding graphics primitives. The next image selects and shows only those triangles in the previous image whose centers have x-coordinates ≤ 0.

```
Show[Graphics3D[
   {#[[1]], Select[#[[2]], (* the selection criteria *)
    (N[First[Total[#[[1]]]/3]] <= 0)&]}& /@ %[[1]]],
     PlotRange -> All, Lighting -> False,
     ViewPoint -> {3, 0, 1}, Boxed -> False]
```

In the next graphic, we manipulate directly the polygons of the above picture.

```
Show[DeleteCases[
Show[%% /. (* invert *)
          Polygon[l_] :> Polygon[#/#.#& /@ l],
          (* split intersecting polygons *)
          PolygonIntersections -> False], _Line, Infinity] /.
          (* shrink resulting polygons *)
     Polygon[l_] :> With[{mp = Mean[l]},
       {EdgeForm[], Polygon[(mp + 0.7(# - mp))& /@ l]}],
     PlotRange -> All, Boxed -> False,
     Lighting -> False, BoxRatios -> {1, 1, 1}]
```

3D graphics can be converted into 2D graphics and the resulting 2D polygons, lines, and points can be further manipulated within *Mathematica*. The following Christmas-themed input generates 413 random polyhedra, projects them into 2D, and places the resulting graphics on a grid in a random order.

```
(* load polyhedra package *)
Needs["Graphics`Polyhedra`"];
```

```
manyRandomPolyhedra[{Lx_, Ly_, δ_}] :=
Module[{randomPolyhedra, randomProjectedPolyhedra,
        randomPermutation, frame, d = Min[Lx, Ly]/20, ε = 0.5},
(* a random polyhedron *)
randomPolyhedra[n_] :=
{SurfaceColor[Hue[Random[]], Hue[Random[]], 3 Random[]],
  (* iterate a random truncation/stellation *)
 Nest[(* random truncation or stellation *)
      If[Random[Integer] === 0,
          Truncate[#, Random[Real, {0.1, 0.4}]],
          Stellate[#, Random[Real, {1.3, 1.9}]]]&,
     (* randomly select a Platonic solid *)
     Polyhedron[{Tetrahedron, Hexahedron,
                 Octahedron, Dodecahedron, Icosahedron}[[
                             Random[Integer, {1, 5}]]]], n][[1]]};
(* project into 2D *)
randomProjectedPolyhedra[mp2D_] := Graphics[
Show[Graphics3D[randomPolyhedra[Random[Integer, {2, 3}]]],
      ViewPoint -> Table[Random[Real, {3/4, 4}], {3}],
      Boxed -> False, DisplayFunction -> Identity,
      PlotRange -> All, SphericalRegion -> True] /.
      (* color each face differently *) p_Polygon :>
      {SurfaceColor[Hue[Random[]], Hue[Random[]], 3 Random[]], p}] /.
   (* center approximately around origin and color lines *)
   (pl:(Polygon | Line))[l_] :> pl[(mp2D + # - {0.4, 0.4})& /@ l] /.
    Graphics[l_] :> Graphics[{Thickness[0.001],
                               GrayLevel[Random[Real, {0.25, 1}]], l}];
(* random permutation of a list *)
randomPermutation[l_] :=
Module[{ℓ = l, n = Length[l]}, Do[(ℓ[[{k, #}]] = ℓ[[{#, k}]])&[
          Random[Integer, {k, n}]], {k, n}]; ℓ];
(* frame *)
frame[{lx_, ly_}, d_] :=
With[{ℓ = {{-1, -1}, {1, -1}, {1, 1}, {-1, 1}, {-1, -1}}},
     Polygon[Join[{lx, ly} #& /@ ℓ, Reverse[{lx + d, ly + d} #& /@ ℓ]]]];
(* centers of projected polyhedron a grid;
  use random order of centers *)
mps = randomPermutation[Flatten[Table[{x, y},
        {x, -Lx, Lx, δ}, {y, -Ly + δ/2 Mod[x/δ, 2], Ly, δ}], 1]];
(* display graphic *)
Show[{(* random projected polyhedra *)
      randomProjectedPolyhedra /@ mps,
      Graphics[{Hue[0], frame[{Lx, Ly}, d]}]},
     AspectRatio -> Automatic, AspectRatio -> Automatic,
     PlotRange -> {(Lx + ε d) {-1, 1}, (Ly + ε d) {-1, 1}}]]

SeedRandom[999];
manyRandomPolyhedra[{4, 3/2, 1/4}]
```

Because of its symbolic, numeric, pattern-matching, and graphical capabilities, constructing a variety of pictures is easy with *Mathematica*. Here are two further variations of a polyhedral flower.

```
Needs["Graphics`Polyhedra`"];
Needs["Graphics`Shapes`"];

With[{pp = 30},
Show[GraphicsArray[{Graphics3D[{
             EdgeForm[{Hue[0.22], Thickness[0.001]}],
             SurfaceColor[Hue[0.3], Hue[0.45], 1.2],
  (* make polygons *)
   Map[MapThread[Polygon[Join[#1,
    Reverse[#2]]]&, #]&, Map[Partition[#, 2, 1]&, Map[
     Partition[#, 2, 1]&, Transpose[Map[First, Table[
    (* rotate faces outwards *)
      RotateShape[Map[Function[l,
   Module[{mp = Mean[l]},
          mp + 0.6(1 - p^2) (# - mp)& /@ l]],
    Map[(p - 1)#&, Line[Append[#, First[#]]]& /@ First /@
   Polyhedron[Dodecahedron][[1]], {-1}], {2}],
     p^2/2, -p^2/2, p^2/2], {p, -1, 1, 2/pp}], {2}]],
      {1}], {3}], {2}]}, Boxed -> False],
 (* form Graphics3D-object *)
 Graphics3D[{EdgeForm[{Hue[0.77], Thickness[0.001]}],
             SurfaceColor[Hue[0.22], Hue[0.85], 1.6],
 (* make polygons *)
  Map[MapThread[Polygon[Join[#1,
   Reverse[#2]]]&, #]&, Map[Partition[#, 2, 1]&, Map[
    Partition[#, 2, 1]&, Transpose[Map[First, Table[
    (* rotate faces outwards *)
     RotateShape[Map[Function[l,
  Module[{mp = Total[l]/3},
         mp + 0.5(1 - p^2) (# - mp)& /@ l]],
   Map[(p - 1)#&, Line[Append[#, First[#]]]& /@ First /@
  Polyhedron[Icosahedron][[1]], {-1}], {2}],
   p^3/2, Sin[Pi p]/2, p/4], {p, -1, 1, 2/pp}], {2}]],
    {1}], {3}], {2}]}, Boxed -> False]}]]]
```

It is possible to visualize real objects by using points, lines, and polygons directly in 3D space. Obtaining “realistic” images usually requires generating a large number of polygons. We could go on and display windmills, torsos, autos, starfish, cathedrals, castles, gears, the Eiffel tower [511✱], the Sagrada Familia, and so on.

We will use more graphics in the next two subsections for various visualizations.

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

■ 1.2.3 Symbolic Calculations

`D[f[x], x]` differentiates $f(x)$ once with respect to x.

```
D[Sin[x], x]
```

Here is a slightly more complicated expression.

```
f = Sin[Log[Tan[(ξ^2 + Exp[x])/(Cos[ξ^2 - 1] + Sqrt[ξ])]]]
```

The resulting manual differentiation is somewhat unpleasant. The result of differentiating this expression twice with respect to ξ is quite big, so we use `Short` to force *Mathematica* to show only a part.

```
D[f, {ξ, 2}] // Short[#, 4]&
```

Here is a simple integral.

```
Integrate[Sin[x], x]
```

The following integral is tedious to find by hand.

```
Integrate[ξ^3 Sin[ξ]^4, ξ]
```

By differentiating and simplifying, we get $\xi^3 \sin(\xi)^4$ again.

```
D[%, ξ]
```

```
Simplify[%]
```

Here is the definite integral $\int_{-\infty}^{\infty} (x^4 + 4)^{-2}\, dx$.

```
Integrate[1/(x^4 + 4)^2, {x, -Infinity, Infinity}]
```

We now consider a function that is complicated for integration.

```
g = t^(2/3) Exp[-2 t] (t - 1)^(4/5)
```

It can be integrated analytically over the domain 1 to ∞.

```
Integrate[g, {t, 1, Infinity}]
```

Because all of the special functions are numerically implemented for arbitrary complex arguments (in their domains) with arbitrary accuracy, we can also compute the numerical value with 50 digits.

```
N[%, 50]
```

Here is the same integral calculated numerically to ten digits.

```
NIntegrate[Evaluate[g], {t, 1, Infinity},
          PrecisionGoal -> 10] // InputForm
```

Here, the function $\sin(x^2)$ is integrated five times.

```
Integrate[Sin[x^2], x, x, x, x, x]
```

Differentiating the result five times brings us back to $\sin(x^2)$.

```
D[%, x, x, x, x, x] // Simplify
```

The next input calculates the even Bernoulli numbers through the integral representation [580✱]

$$B_{2n} = -(-1)^m\, 2^{-(2m+1)} \int_{-\infty}^{\infty} \left(\frac{d^{m-1}\, \operatorname{sech}^2(x)}{d\, x^{m-1}} \right)^2 .$$

```
Table[-(-1)^n 2^-(2n + 1) Integrate[D[Sech[x]^2, {x, n - 1}]^2,
                                    {x, -Infinity, Infinity}],
     {n, 10}]
```

The Bernoulli numbers are built-in functions of *Mathematica*.

```
Table[BernoulliB[2n], {n, 10}]
```

Now let us consider a limit. The function $e^{1/(x-1)}$ has two different limit values, one from the left and from the right at the point $x = 1$.

```
Limit[Exp[-1/(1 - x)], x -> 1, Direction -> +1]
```

```
Limit[Exp[-1/(1 - x)], x -> 1, Direction -> -1]
```

We now solve a differential equation describing a damped oscillation $x''(t) + \gamma\, x'(t) + \omega^2\, x(t) = 0$.

```
DSolve[x''[t] + γ x'[t] + ω^2 x[t] == 0, x[t], t]
```

Suppose we want to approximate a function $f(x)$ with the following properties by a polynomial in x:

$$\begin{aligned} f(0) &= 1 \\ f'(0) &= 2 \\ f(4) &= 8 \\ f'(4) &= 45 \\ f''(4) &= 0. \end{aligned}$$

```
InterpolatingPolynomial[{{0, {1, 2}}, {4, {8, 45, 0}}}, x]
```

Here is the same polynomial in a simpler, but less practical, form.

```
Simplify[%]
```

We check that it interpolates.

```
{% /. {x -> 0}, D[%, x] /. {x -> 0}, % /. {x -> 4},
 D[%, x] /. {x -> 4}, D[%, {x, 2}] /. {x -> 4}}
```

Here is the piecewise-continuous function $\mathrm{pw}(x) = \theta(x)\,\theta(1-x)\,\lfloor 5x^6 - 1\rfloor\,\mathrm{frac}(x^4-4)^3\,\left(\left|\lceil x^2+2x-1\rceil^6\right|\right)^{1/3}$ defined.

```
pw[x_] = If[0 < x < 1, Floor[(5 x^6 - 1)] FractionalPart[x^4 - 4]^3*
                       Abs[Ceiling[x^2 + 2 x - 1]^6]^(1/3), 0]
```

Here is a canonical form of this function.

```
PiecewiseExpand[pw[x]]
```

The next input calculates $\int_{-\infty}^{\infty}\mathrm{pw}(x)^2\,dx$.

```
Integrate[pw[x]^2, {x, -Infinity, Infinity}]
```

And here is a series expansion of this function at a point where a discontinuity occurs.

```
Series[pw[x], {x, Sqrt[3] - 1, 1}]
```

Next, we solve a well known-differential equation of mathematical physics describing (among other things) the behavior of a quantum particle in a constant electric field.

```
DSolve[ψ''[z] + e F z ψ[z] == ψ[z], ψ[z], z]
```

The Vandermonde matrix of the nth-order is easy to implement in the following way.

```
VandermondMatrix[n_, x_] := Table[x[i]^j, {i, 0, n}, {j, 0, n}]
```

Here is the Vandermonde matrix of the third order. `x[i]` is a typical *Mathematica* equivalent for x_i.

```
MatrixForm[VandermondMatrix[3, x]]
```

Here is the value of its determinant.

```
Det[VandermondMatrix[3, x]]
```

This product can also be written as a product.

```
Factor[%]
```

The following function `LUMatricesVandermonde` implements the LU-decomposition of the nth-order Vandermonde matrix [1401✱].

```
LUMatricesVandermonde[n_, x_] :=
Module[{X = Table[x[k], {k, 0, n}], e, h, b, L, U},
  (* recursive definitions for elementary and complete
   symmetric polynomials *)
 e[0, _] := 1; e[_, {}] := 0; h[0, _] := 1; h[_, {}] := 0;
 e[r_, l_] := Factor[e[r, Most[l]] + Last[l] e[r - 1, Most[l]]];
 h[r_, l_] := Factor[h[r, Most[l]] + Last[l] h[r - 1, l]];
 b[r_, y_] := Factor[Sum[(-1)^(r - k) e[r - k, Take[X, r]] y^k, {k, 0, r}]]
 (* lower and upper triangular matrices *)
 L = Table[If[i < j, 0, h[i - j, Take[X, j + 1]]], {i, 0, n}, {j, 0, n}];
 U = Table[If[i > j, 0, b[i, X[[j + 1]]]], {i, 0, n}, {j, 0, n}];
 (* return matrices *)  {L, U}]
```

Here is the decomposition for the third order Vandermonde matrix.

```
{L, U} = LUMatricesVandermonde[3, x];
{L, U} // (MatrixForm /@ #)&
```

Multiplying the two matrices recovers the original matrix.

```
L.U // Expand
```

The last LU-decomposition is not unique. The next inputs use the function `Solve` to calculate the most general solution.

```
(* general ansatz form for the matrices L and U *)
With[{n = 3}, MatrixForm /@
 {L = Table[If[i < j, 0, l[i, j]], {i, 0, n}, {j, 0, n}],
  U = Table[If[i > j, 0, u[i, j]], {i, 0, n}, {j, 0, n}]}]
```

```
Solve[(* the decomposition identity that must hold *)
      L.U == VandermondMatrix[3, x],
      (* the 10 variables l[i, j] and the 10 variables u[i, j] *)
      Cases[{L, U}, _l | _u, Infinity]] // (Factor //@ #)&
```

Next, we calculate symbolically the eigenvalues of a 50×50 Redheffer matrix. The matrix elements a_{ij} are 1 if $j = 1$ or if i divides j, and 0 otherwise.

```
Redheffer𝔸[d_] := Table[If[j == 1 || IntegerQ[j/i], 1, 0], {i, d}, {j, d}];
```

A Redheffer matrix of dimension n has $n - \lfloor \log_2(n) \rfloor - 1$ eigenvalues 1 (see [1312✱] and [1313✱]). The remaining six (for $n = 50$) eigenvalues are the roots of an irreducible polynomial of degree 6. They are represented as `Root`-objects.

```
Eigenvalues[Redheffer𝔸[50]]
```

The following example is a linear inhomogeneous system of equations with eight unknowns. (We show the equations in abbreviated form.)

```
gls = Table[Sum[(i + j)^j x[i], {i, 8}] == j, {j, 8}];

Short /@ gls
```

We get its exact solution.

```
Solve[gls, Table[x[i], {i, 8}]]
```

Here is a simple system of nonlinear equations and its solution.

$$x^2 + y^2 = 1$$
$$x^4 + y^4 = 4$$

```
Solve[{x^2 + y^2 == 1, x^4 + y^4 == 4}, {x, y}]
```

The function `Eliminate` eliminates variables from a system of polynomial equations.

```
Eliminate[{x^6 + y^6 == 6, x^8 + y^8 == 8}, {y}]
```

Mathematica can also solve higher order univariate polynomial equations.

```
Solve[x^7 - a x + 3 == 0, x]
```

The result contained again the `Root` function. `Root`-objects are symbolic representations of the roots of polynomials. The first argument specifies the polynomial, and the second, the root number. (See Chapter 1 of the Symbolics volume of the *GuideBooks* [1285✱] for details.) Like any other function in *Mathematica,* they can be manipulated, for instance, differentiated.

```
D[Root[-3 + a #1 - #1^7 &, 1], {a, 2}]
```

Here is the numerical value of the root for a given value of `a` to 50 digits.

```
N[Root[-3 + a #1 - #1^7 &, 1] /. a -> 7, 50]
```

Backsubstitution shows that the equation gives zero to a good approximation.

```
x^7 - a x + 3 /. a -> 7 /. x -> %
```

Here is a plot of the root; the parameter `a` varies between -10 and 10.

```
Plot[Root[-3 + a #1 - #1^7 &, 1], {a, -10, 10}]
```

The following input solves a transcendental equation.

```
Solve[Log[2 x] + Log[3 x] + Log[5 x] == 1/2, x]
```

The following example is a simple power series expansion up to the ninth order.

```
Series[Sqrt[1 + x], {x, 0, 9}]
```

The next one is not so simple. It is not a Taylor series because logarithms appear.

```
Series[x^x, {x, 0, 4}]
```

What is the first nonvanishing term in the series expansion of $\sin(\tan(x)) - \tan(\sin(x))$ [1235✱]?

```
Series[Sin[Tan[x]] - Tan[Sin[x]], {x, 0, 9}]
```

Here is a Laurent series.

```
Series[1/(Sin[x] - x - x^3/3 + x^5), {x, 0, 6}]
```

Here is a short program using l'Hôspital's rule for determining the limit of $\frac{\sin(\tan(x))-\tan(\sin(x))}{\arcsin(\arctan(x))-\arctan(\arcsin(x))}$ as $x \to 0$.

```
numerator = Sin[Tan[x]] - Tan[Sin[x]];
denominator = ArcSin[ArcTan[x]] - ArcTan[ArcSin[x]];
```

We differentiate the numerator and denominator until we get a determined quantity.

```
lHospitalList = Table[D[numerator, {x, i}]/D[denominator, {x, i}], {i, 7}];

If[(* zero denominator? *) (Denominator[#] /. x -> 0) == 0,
    Indeterminate, # /. x -> 0]& /@ lHospitalList
```

Of course, *Mathematica* can also calculate this limit directly. (*Mathematica* can also compute limits in cases in which l'Hôspital's rule is not applicable [586✱], [163✱], [1131✱].)

```
Limit[(Sin[Tan[x]] - Tan[Sin[x]])/
      (ArcSin[ArcTan[x]] - ArcTan[ArcSin[x]]), x -> 0]
```

Here is an exact value for the Gauss hypergeometric function with numeric arguments.

```
Hypergeometric2F1[3/2, 4, 1, z]
```

We can also evaluate a high-order Hermite polynomial.

```
HermiteH[23, z]
```

The command `FunctionExpand` rewrites an expression using a simpler function than the original one. In the following, a trigonometric expression is converted to one involving square roots only.

```
FunctionExpand[Sin[1/(2^3 3 5) Pi]]
```

Here is a similar example.

```
FunctionExpand[Tan[Pi/32]]
```

The previous expression is an algebraic number. It is a root of the polynomial that is the first argument of the following `Root`-object.

```
RootReduce[%]
```

Next, we find the prime factor decomposition of a relatively large number.

```
FactorInteger[4951486756871515]
```

Here is an abbreviated list of all numbers dividing the number 4951486756871515.

```
Divisors[4951486756871515] // Short[#, 6]&
```

This is the one-billionth prime number.

```
Prime[10^9]
```

We now decompose a polynomial into smaller ones that, when plugged into each other, give again the starting polynomial. (This specific example was already decomposed by Vieta in 1594 [232✱].)

```
Decompose[45 x - 3795 x^3 + 95634 x^5 - 1138500 x^7 + 7811375 x^9 -
          34512075 x^11 + 105306075 x^13 - 232676280 x^15 +
          384942375 x^17 - 488494125 x^19 + 483841800 x^21 -
          378658800 x^23 + 236030652 x^25 - 117679100 x^27 +
          46955700 x^29 - 14945040 x^31 + 3764565 x^33 -
          740259 x^35 + 111150 x^37 - 12300 x^39 + 945 x^41 -
          45 x^43 + x^45, x]
```

Sums can also be computed symbolically. Here are the first few partial sums for $\sum_{k=1}^{n} k^j$.

```
TableForm[Table[Sum[k^j, {k, n}], {j, 1, 8}]]
```

It is even possible to compute infinite sums analytically.

```
Sum[1/k^6, {k, Infinity}]
```

Here are two more complicated sums. The summands and the result contain Riemann's Zeta function.

```
Sum[(-1)^n/n^2 Gamma[n]^2/Gamma[2n], {n, Infinity}]

Sum[(Zeta[k] - 1) Exp[-k], {k, 2, Infinity}]
```

Here is a complicated finite sum. The result contains the Polygamma function and a sum of roots of a quartic polynomial.

```
Sum[(k^2 - 1)/(k^4  + 1), {k, 1, n}]
```

Mathematica's functions `Integrate`, `Sum`, `DSolve` are very powerful and can integrate, sum, and solve differential equations of quite complicated functions. However, for efficiency, the solution is typically not automatically simplified. But *Mathematica* provides a variety of functions allowing us to rewrite results from functions like `Integrate`, `Sum`, `DSolve` in various ways. For instance, here is a more explicit form of the last result (not containing the function `RootSum` any more).

```
Normal[%] // Simplify
```

Using the function `FullSimplify`, we can further collapse the last result.

```
% // FullSimplify
```

A closed form for the partial sum of the first n Taylor coefficients of sin(x).

```
Sum[(-1)^k/(2k + 1)! x^(2k + 1), {k, 0, n}] // FullSimplify
```

For a given value of n, we recover the first n Taylor coefficients of sin(x).

```
Series[% /. n -> 12, {x, 0, 12}]
```

Here is a complicated finite sum that can be expressed in polylogarithmic and Lerch functions:

$$\sum_{k=1}^{n} k^{-1/2} (k\,\omega)^{i\,\omega} \left(\frac{x}{k}\right)^{m} x^{2\,i\,\pi\,\gamma\,k}.$$

```
f[{m_, ω_, γ_}, n_, x_] :=
   Sum[k^(-1/2) (k ω)^(I ω) (x/k)^m x^(I 2Pi γ k), {k, n}];

f[{m, ω, γ}, n, x] // PowerExpand // TraditionalForm
```

For larger n the last sums shows a complicated, hierarchical behavior [537✱]. Here is an example for the parameter values $m = 0.2$, $\omega = 7$, $\gamma = 1.007$, and $n = 100$.

```
Plot[Evaluate[Im[f[{0.2, 7, 1.007}, 100, x]]], {x, 0, 1},
     PlotPoints -> 1000, Frame -> True, Axes -> False]
```

The following sum calculates the interaction energy of a point charge at position z_0 between two flat, parallel, perfectly conducting walls of distance a using mirror charges [1192✱].

```
Sum[1/(a n) - 1/(2n a - 2 z0) - 1/(2 (n - 1) a + 2 z0),
    {n, Infinity}] // Normal // Simplify
```

A Green's function approach to the same problem yields the following integral and, of course, evaluates to the same result [1192✱].

```
Integrate[Cosh[k a]/Sinh[k a] - 1 - Cosh[k(a - 2 z0)]/Sinh[k a],
          {k, 0, Infinity},
          Assumptions -> a > 0 && z0 > 0 && z0/a < 1]
```

A series expansion of the energy around $z_0 = 0$ or $z_0 = a$ yields the force on the point charge.

```
{Series[%, {z0, 0, 6}], Series[%, {z0, a, 6}]} // FullSimplify
```

Next, we use *Mathematica* to prove a neat identity discovered by Ramanujan:

$$\sqrt[3]{\cos\left(\frac{2\pi}{9}\right)} + \sqrt[3]{\cos\left(\frac{4\pi}{9}\right)} - \sqrt[3]{\cos\left(\frac{\pi}{9}\right)} = \sqrt[3]{\frac{3\sqrt[3]{9}}{2} - 3}\,.$$

```
Cos[2Pi/9]^(1/3) + Cos[4Pi/9]^(1/3) -
(Cos[1Pi/9])^(1/3) - (3 9^(1/3)/2 - 3)^(1/3)
```

The last identity contains algebraic and trigonometric expressions. For algorithmic treatments, algebraic expressions are always preferable. In algebraic form, the identity has the following form.

```
Together[TrigToExp[%]]
```

The function `RootReduce` canonicalizes algebraic expressions. The identity can be simplified to 0.

```
RootReduce[%]
```

Here is more challenging example: A three-line proof of Legendre's celebrated identity for complete elliptic integrals [442✱]

$$E(m)\,K(1-m) - K(m)\,K(1-m) + E(1-m)\,K(m) = \frac{\pi}{2}.$$

The integral

$$\int_0^{\frac{\pi}{2}}\int_0^{\frac{\pi}{2}} \frac{1 - m\sin^2(x) - (1-m)\sin^2(y)}{\sqrt{1 - m\sin^2(x)}\,\sqrt{1-(1-m)\sin^2(y)}}\,dx\,dy$$

is the left-hand side of Legendre's identity.

```
integrand[x_, y_, m_] := (1  - m Sin[x]^2 - (1 - m) Sin[y]^2)/
          Sqrt[1 - m Sin[x]^2]/Sqrt[1 - (1 - m) Sin[y]^2]

Integrate[integrand[x, y, m], {y, 0, Pi/2}, {x, 0, Pi/2},
          GenerateConditions -> False] // FunctionExpand // Together
```

Differentiation shows that the last expression is independent of m.

```
D[%, m] // Together
```

This means $E(m)\,K(1-m) - K(m)\,K(1-m) + E(1-m)\,K(m)$ equals a constant, and evaluating the above integrand for $m = 0$ shows that the constant is $\pi/2$.

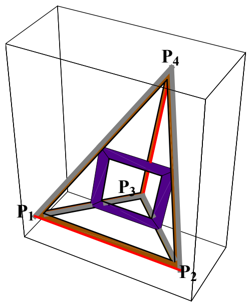
```
Integrate[integrand[x, y, 0], {x, 0, Pi/2}, {y, 0, Pi/2}]
```

A powerful command for algebraic computations is `GroebnerBasis`. Given a set of polynomials, the function `GroebnerBasis` can transform this set into triangular form, so that a numerical solution is easily possible. `Groeb⋱ nerBasis` can also be used to eliminate certain variables from a set of polynomials. In the following example, we are looking for an equation connecting the area A of a triangle with the radius of its circumscribed circle, with radius R and the edge lengths l_{12}, l_{13}, and l_{23}.

```
Clear[x1, y1, x2, y2, x3, y3, X, Y, R, A];
GroebnerBasis[{(* all equations of the problem *)
  (* defining equations for the circumscribed circle *)
   (X - x1)^2 + (Y - y1)^2 - R^2,
   (X - x2)^2 + (Y - y2)^2 - R^2,
   (X - x3)^2 + (Y - y3)^2 - R^2,
  (* defining equations for the length of the edges *)
   (x2 - x1)^2 + (y2 - y1)^2 - l12^2,
   (x3 - x2)^2 + (y3 - y2)^2 - l32^2,
   (x1 - x3)^2 + (y1 - y3)^2 - l13^2,
  (* defining equations for area *)
   (1/2(-x2 y1 + x3 y1 + x1 y2 - x3 y2 - x1 y3 + x2 y3))^2 - A^2},
  (* the variables to keep *) {R, l12, l23, l13, A},
  (* the variables to eliminate *) {x1, y1, x2, y2, x3, y3, X, Y}]
```

The last polynomial in the result means that the relation we were looking for is $A = l_{12}\,l_{23}\,l_{13}/(4\,R)$. The first polynomial in the result expresses the area in the edge lengths only.

Let us use `GroebnerBasis` [288✱] again to solve a slightly more complicated example: the area of the medial parallelogram [28✱] of a tetrahedron expressed through the edge length of the tetrahedron [1404✱], [65✱]. We start with a generic tetrahedron. From this tetrahedron, we remove two nonincident edges. The midpoints of the remaining four edges form a parallelogram, the medial parallelogram. We want to express the area of this parallelogram through the six lengths of the edges of the original tetrahedron. Below is a sketch of the tetrahedron. The two red-colored edges $\overline{P_1 P_2}$ and $\overline{P_3 P_4}$ are the removed edges.



The next input calculates the formula we are looking for.

```
Module[{(* coordinates of the four vertices *)
        p1 = {0, 0, 0}, p2 = {p2x, 0, 0},
        p3 = {p3x, p3y, 0}, p4 = {p4x, p4y, p4z},
        p13, p14, p23, p24},
  (* coordinates of the midpoints of the edges *)
  p13 = (p1 + p3)/2; p14 = (p1 - p4)/2;
  p23 = (p2 + p3)/2; p24 = (p2 + p4)/2;
 GroebnerBasis[
    {(* edge lengths expressed through coordinates of vertices *)
     l12^2 - (p1 - p2).(p1 - p2), l13^2 - (p1 - p3).(p1 - p3),
     l23^2 - (p2 - p3).(p2 - p3), l14^2 - (p1 - p4).(p1 - p4),
     l24^2 - (p2 - p4).(p2 - p4), l34^2 - (p3 - p4).(p3 - p4),
     (* medial parallelogram area A expressed
        through coordinates of vertices *)
     A^2 - Cross[p14 - p13, p23 - p13].Cross[p14 - p13, p23 - p13]},
     (* the variables to keep *)
     {l12, l13, l14, l23, l24, l34, A},
     (* the variables to eliminate *)
     {p2x, p3x, p3y, p4x, p4y, p4z}, MonomialOrder -> EliminationOrder]]
```

In the last subsection, we made use of the `Polyhedra`` package. *Mathematica* comes with a wide set of standard packages carrying out various numerical, graphical, and symbolic operations not built into the *Mathematica* kernel. Let us make use of the package `Algebra`InequalitySolve`` for doing some symbolic calculations. The package implements the function `InequalitySolve`.

```
Needs["Algebra`InequalitySolve`"]
```

As the function name indicates, `InequalitySolve` "solves" inequalities. Solving an inequality here means describing the solution sets in a canonicalized manner. The canonicalized form is a hierarchical description of the allowed intervals for the variables.

```
?InequalitySolve
```

Next, we "solve" the inequality $\left(-16\,x^6+24\,x^4-9\,x^2-4\,y^4+4\,y^2\right)^2-1/8<0$.

```
𝕃[x_, y_] = (24x^4 - 9x^2 - 16x^6 + 4y^2 - 4y^4)^2 - 1/8;
```

```
iSol = InequalitySolve[𝕃[x, y] < 0, {x, y}];
```

Because the result `iSol` is quite large and its structure is not immediately recognizable, we do not display the result. It has 25 independent parts.

```
iSol // Length
```

Here is the first part.

```
First[iSol]
```

It is of the form $x_1 < x < x_2 \bigwedge \left(y_1(x) < y < y_2(x) \bigvee \tilde{y}_1(x) < y < \tilde{y}_2(x)\right)$. This form is the canonicalized description of one region where the above inequality holds. The regions are areas or lines extending along the y-direction over a fixed x-interval. (For a more detailed description, see Section 1.2.3 of the Symbolics volume [1285✱] of the *GuideBooks*.) Similar to the above `Solve` example, when “solving” inequalities, one often ends up with `Root`-objects. The x_1, x_2 are algebraic numbers and $y_1(x)$, $y_2(x)$, $\tilde{y}_1(x)$, and $\tilde{y}_2(x)$ are algebraic functions of x_1 and x_2, which means they are inverse functions of polynomials that generically cannot be inverted using elementary functions.

It is straightforward to visualize the canonicalized regions where the inequality holds. We just form polygons by traversing $y_1(x)$ from x_1 to x_2 and going back along $y_2(x)$ from x_2 to x_1 and similarly for $\tilde{y}_1(x)$, $\tilde{y}_2(x)$. The little function `makePolygon` forms a polygon from a logical combination of inequalities.

```
makePolygon[Inequality[x1_, Less, x, Less, x2_] &&
            Inequality[y1_, Less, y, Less, y2_],
            plotpoints:pp_Integer] :=
With[{(* avoid endpoints *) ε = 10.^-12}, Polygon[Join[
  (* bottom and top boundaries *)
 Table[{x, y1}, {x, x1 + ε, x2 - ε, (x2 - x1 - 2ε)/pp}],
 Table[{x, y2}, {x, x2 - ε, x1 + ε, (x1 - x2 + 2ε)/pp}]]]]
```

`iSol` contains 41 independent 2D regions. Here, we show them; each one has a randomly assigned color. (The regions described by the inequality are “thickened” versions of the Lissajous curve $\{x,\ y\} = \{\sin(2\,\vartheta), \cos(3\,\vartheta)\}$. As a guide for the eye, we display this curve in gray on top of the colored regions.)

```
Show[{Graphics[{Thickness[0.01],
     {Hue[Random[]], makePolygon[#, 20]}& /@
       (* ignore one-dimensional parts *)
      Apply[List, (DeleteCases[iSol, _Equal && _] /.
                   a_ && b_Or :> ((a && #)& /@ b))]}],
     (* the Lissajous curve *)
     ParametricPlot[{Sin[2ϑ], Cos[3ϑ]}, {ϑ, 0, 2Pi},
              PlotRange -> All, PlotPoints -> 200,
              DisplayFunction -> Identity,
              PlotStyle -> {{GrayLevel[0.5], Thickness[0.01]}}]},
     AspectRatio -> Automatic, Frame -> True,
     PlotRange -> {{-1.2, 1.2}, {-1.2, 1.2}}]
```

The next input finds the smallest value of R, such that all points $\{\xi,\ \eta\}$ with $|\{\xi,\ \eta\}| > R$, the value of the left-hand side of the above inequality is positive (meaning the maximal spatial extension of the set defined by the inequality from the origin).

```
ForAll[{ξ, η}, {ξ, η, R} ∈ Reals && Norm[{ξ, η}] > R, 𝕃[ξ, η] > 0] //
                                (* write in quantifier-free form *) Resolve
```

The resulting value of R is a root of a polynomial of degree 15 with integer coefficients. Its numerical value is 1.38143….

```
N[%]
```

The next input proves for positive a, b, c the so-called Nesbitt inequality $a/(b+c) + b/(a+c) + c/(a+b) \geq 3/2$ [1244✱]. We specify the inequality using the forall quantifier.

```
ForAll[{a, b, c}, Element[{a, b, c}, Reals] && a > 0 && b > 0 && c > 0,
       a/(b + c) + b/(a + c) + c/(a + b) >= 3/2] // Resolve
```

In case the constant 3/2 were not known in advance, one could easily determine it, either through quantifier elimination or minimization.

```
ForAll[{a, b, c}, Element[{a, b, c, R}, Reals] && a > 0 && b > 0 && c > 0,
       a/(b + c) + b/(a + c) + c/(a + b) >= R] // Resolve
```

```
Minimize[a/(b + c) + b/(a + c) + c/(a + b),
         a > 0 && b > 0 && c > 0, {a, b, c}]
```

While being inherently of algebraic nature, functions like `Resultant` and `GroebnerBasis` can often be fruitfully used to deal with analysis problems (as we will do repeatedly in the *GuideBooks*). Here we use them to derive nonlinear polynomial differential equations for the function $\mathcal{Y}(z) = \tan(\ln(z))$. Differentiating $\mathcal{Y}(z)$ repeatedly shows powers of the $\sec(\ln(z))$ and $\mathcal{Y}(z)$.

```
Table[Derivative[k][𝒴][z] - D[Tan[Log[z]], {z, k}], {k, 0, 3}] //
                                                 Together // Numerator
```

Eliminating $\sec(\ln(z))^n$ and $\tan(\ln(z))^m$ yields polynomial differential equations such as $z\,\mathcal{Y}''(z) = \mathcal{Y}'(z)\,(2\,\mathcal{Y}(z) - 1)$ in z, $\mathcal{Y}(z)$, $\mathcal{Y}'(z)$, $\mathcal{Y}''(z)$ and maybe higher derivatives of $\mathcal{Y}(z)$.

```
GroebnerBasis[%, {}, {Tan[Log[z]], Sec[Log[z]]},
              MonomialOrder -> EliminationOrder] // Factor
```

Taking two such differential equations yields a z-free, nonlinear, third-order differential equation for $\mathcal{Y}(z) = \tan(\ln(z))$.

```
Resultant[%[[1, -1]], %[[2, -1]], z] // Simplify
```

Substituting $\tan(\ln(z))$ for $\mathcal{Y}(z)$ in the last differential equations gives zero.

```
% /. {𝒴[z] :> Tan[Log[z]],
      Derivative[k_][𝒴][z] :> D[Tan[Log[z]], {z, k}]} // Simplify
```

Next, we examine a self-defined rule. The function $x^p \sin(x^q) \ln(x^r)$ cannot be integrated by *Mathematica* with respect to x (it is not possible to express this integral in named special functions).

```
Integrate[x^p Tan[x^q] Log[x^r], {x, 0, Pi}]
```

However, we can create a new symbol `XtoPowerαTimesSinOfXtoPowerβTimesLogOfXtoPowerγ[p, q, r]` for this integral.

```
Unprotect[Integrate];

Integrate[x_^α_. Tan[x_^β_.] Log[x_^γ_.], {x_, 0, Pi}] :=
          XtoPowerαTimesTanOfXtoPowerβTimesLogOfXtoPowerγ[α, β, γ];

Protect[Integrate];
```

Mathematica can use this rule when it is possible.

```
Integrate[z^I Tan[z^23] Log[z], {z, 0, Pi}]
```

Mathematica is good at matching patterns. For example, we can extract all elements from a list that are the product of `x` with any factor, including the not explicitly written factor 1.

```
Cases[{3, 2 + 7 I, 6 x, I x, u x, x, a x, u}, Optional[_] x]
```

Here is an umbral example [380✱]. When one interprets the even powers $\mathcal{E}^k$ in the expanded form of $(\mathcal{E} - i)^n = 0$ as indexed numbers $\mathcal{E}_k$, then the $\mathcal{E}_k$ are just the absolute values of the Euler numbers $|E_k|$ [439✱], [479✱]. Here is an example for $n = 12$. This is the expanded form.

```
Expand[(ℰ - Sqrt[-1])^12]
```

Using patterns and replacements is straightforward to go from the monomials $\mathcal{E}^k$ to the indexed quantities $\mathcal{E}_k$.

```
% /. ℰ^k_. :> Subscript[ℰ, k]
```

This checks the above statement about the Euler numbers.

```
% /. Subscript[ℰ, k_] :> Abs[EulerE[k]]
```

The next input tests if the first four digits of π appear somewhere within the first 50000 digits of the decimal representation of 17^{-1000}. (It turns out that within the periodic part of the decimal expansion of 17^{-1000}, the first 1230 digits of π appear many times [1251✱], [1252✱]; almost all real numbers are lexicons [226✱], [584✱].)

```
MatchQ[First[RealDigits[N[1/17^1000, 50000]]], {___, 3, 1, 4, 1, ___}]
```

The first four digits of π appear also in the (integer) digits of 17^{1000}.

```
MatchQ[IntegerDigits[17^1000], {___, 3, 1, 4, 1, ___}]
```

Mathematica can simplify expressions when it knows properties of the variables. In the next input, it is assumed that p is an odd prime.

```
Simplify[Sin[p^2 Pi] + (-1)^p, Element[p, Primes] && p > 2]
```

The following expression does not automatically "simplify" to $x + 1$.

```
Sqrt[x^2 + 2 x + 1]
```

Actually, such a transformation would be mathematically wrong for many complex numbers.

```
{Sqrt[x^2 + 2 x + 1], x + 1} /. x -> -3 + 2I
```

Under the additional assumption that x is a positive real number, *Mathematica* can simplify $\sqrt{x^2 + 2x + 1}$ to $x + 1$.

```
Simplify[Sqrt[x^2 + 2 x + 1], Element[x, Reals] && x > 0]
```

Many more functions in *Mathematica* perform symbolic mathematics. The Numerics [1284✱] and Symbolics [1285✱] volumes of the *GuideBooks* discuss many more details.

Mathematica can carry out complicated and never-before-carried out calculations in various mathematical topics with great ease. The following short code, for instance, searches for a number whose digits of its decimal expansion digits agree with the terms of its (nonsimple) continued fraction expansion.

```
(* difference between decimal expansion and continued fraction *)
δ[l_] := N[Abs[FromDigits[{l, 1}, 10] -
             Fold[#2[[2]]/(#2[[1]] + #1)&, l[[-2]]/l[[-1]],
                       Partition[Reverse[Drop[l, -2]], 2]]]];

(* recursively add digit pair and keep a set of best lists  *)
Nest[First /@ Take[#, Min[43, Length[#]]]&[Sort[{#, δ[#]}& /@
          Flatten[Flatten[Table[Join[#, {i, j}],
                       {i, 0, 9}, {j, 9}], 1]& /@ #, 1],
            (#1[[2]] < #2[[2]])&]]&, {{0}}, 72][[1]]
```

After running, the code above returns the following result.

```
{0, 2, 7, 3, 9, 4, 4, 1, 9, 5, 7, 3, 9, 2, 7, 1, 6, 1,
 7, 1, 7, 1, 4, 5, 9, 1, 5, 2, 7, 2, 4, 2, 8, 5, 9, 1,
 9, 2, 7, 3, 7, 2, 5, 1, 8, 7, 7, 2, 9, 8, 8, 1, 9, 8,
 6, 2, 9, 1, 9, 1, 7, 3, 8, 3, 7, 5, 5, 2, 8, 1, 7, 1,
 7, 7, 4, 1, 8, 1, 9, 6, 9, 4, 6, 1, 9, 1, 7, 3, 8, 2,
 8, 3, 6, 2, 5, 1, 6, 1, 5, 4, 8, 5, 9, 3, 6, 4, 7, 1,
 9, 2, 5, 8, 9, 4, 9, 8, 9, 1, 5, 1, 7, 2, 7, 3, 9, 1,
 9, 6, 7, 6, 9, 2, 8, 1, 9, 4, 5, 3, 5, 1, 6, 3, 8, 1, 6};
```

The next input forms the continued fraction corresponding to the last list.

```
With[{ℓ = C /@ %},
DeleteCases[Hold[0 + #]& @@ {Fold[#2[[2]]/(#2[[1]] + #1)&,
             ℓ[[-2]]/ℓ[[-1]], Partition[Reverse[Drop[ℓ, -2]], 2]]},
     C, Infinity, Heads -> True]] // InputForm
```

Collapsing the last expression into a fraction and then calculating a high-precision approximation of this fraction yields a decimal number, showing that the first 100 digits agree with the continued fraction terms.

```
ReleaseHold[%]
```

```
N[%, 100]
```

Here is a short way to show the agreement of the first 100 digits using *Mathematica*.

```
RealDigits[%%, 10, 100, 0][[1]] == Take[%%%%, 100]
```

The code above can easily be adapted to calculate numbers with many identical decimal and continued fraction digits, for the case of a simple continued fraction, and to deal with the case for a base different from 10.

Mathematica also allows larger mathematical formulas and algorithms to be entered in a direct way. As a small example, let us implement the calculation of the series of the conformal map $w = f(z)$ after Szegö's method (see [346✱], [509✱], [1106✱], [702✱], and [1180✱]), which maps a square in the z-plane onto the unit disk in the w-plane. The approximation of $w = f(z)$ of order n is given by:

$$h_{jk} = \frac{1}{\lambda}\int_C z^j\,\bar{z}^{\,k}\,ds$$

$$\mathbf{H}^{(n)} = h_{jk} \quad j, k = 0, 1, \ldots, n$$

$$d_n = \det \mathbf{H}^{(n)}$$

$$\mathbf{G}^{(n)}(\xi) = \begin{cases} h_{jk}, & j = 0, 1, \ldots, n,\ k = 0, 1, \ldots, n-1 \\ \xi^j, & j = 0, 1, \ldots, n,\ k = n \end{cases}$$

$$l_n(\xi) = \det \mathbf{G}^{(n)}(\xi)$$

$$p_n(\xi) = \frac{l_n(\xi)}{\sqrt{d_{n-1}\,d_n}}$$

$$p_0(\xi) = 1$$

$$k_n(\alpha, \beta) = \sum_{i=0}^{n} p_i(\alpha)\,p_i(\beta)$$

$$w_n(z) = \frac{\pi}{4\,k_n(0, 0)}\int_0^z k_n(0, \xi)^2\,d\xi$$

Here, λ is the length of the boundary of the square, and the integration has to be carried out along the boundary of the square. The $p_n(\xi)$ form orthogonal polynomials. $\mathbf{H}^{(n)}$ and $\mathbf{G}^{(n)}$ are square matrices of dimension n with elements $h_{j,k}$, and $g_{j,k}$ respectively.

Here, the above-described method is implemented. *ord* determines the order in z.

```
ConformalMapSquareToUnitDisk[ord_, z_] :=
Module[{h, H, G, d, l, p, k, t, a, b, λ, integrand,
        edgeList = {-1 + I, 1 + I, 1 - I, -1 - I}},
 lineSegments = Partition[Append[edgeList, First[edgeList]], 2, 1];
 (* edge length *)
 λ = Total[Abs[#[[2]] - #[[1]]]& /@ lineSegments];
 (* the h-integrals *)
 integrand[j_, k_] = Plus @@ ((Abs[#[[2]] - #[[1]]]*
 (#[[1]] + t (#[[2]] - #[[1]]))^j*((#[[1]] + t (#[[2]] - #[[1]]))^k /.
                 c_Complex :> Conjugate[c]))& /@ lineSegments);
 (* scalar product *)
 h[j_, k_] := h[j, k] = 1/λ Integrate[integrand[j, k], {t, 0, 1}];
 (* Hankel-Hadamard-Gram determinants *)
 H[n_] := Array[h, {n + 1, n + 1}, 0];
 d[n_] := d[n] = Det[H[n]];
 G[n_, ξ_] := Array[If[#2 < n, h[#1, #2], ξ^#1]&, {n + 1, n + 1}, 0];
 l[n_, ξ_] := l[n, ξ] = Det[G[n, ξ]];
(* Szegö polynomials *)
 p[0, ξ_] = 1;
 p[n_, ξ_] := p[n, x] = l[n, ξ]/Sqrt[d[n] d[n - 1]];
 (* Szegö kernel *)
 k[a_, b_] = Sum[p[i, a] p[i, b], {i, 0, ord}];
 Cancel[Pi/(4 k[0, 0]) Expand[Integrate[k[0, ξ]^2, {ξ, 0, z}]]]]
```

Here is an example (for *ord* = 8, the constant term deviates about 0.09 % from its exact value).

```
ConformalMapSquareToUnitDisk[8, z]
```

Using *Mathematica*'s graphics capabilities, we can easily visualize the conformal map generated by the last function. The left picture shows a mesh in the square with the corners $-1+i$, $1+i$, $1-i$, $-1-i$, and the right picture shows the mesh after mapping; the unit disk is shown underlying in gray.

```
With[{pp = 15},
Module[{points, opts},
 (* points forming the grid *)
 points = Table[N[x + I y], {x, -1, 1, 1/pp}, {y, -1, 1, 1/pp}];
 (* common graphics options *)
 opts[label_] := Sequence[AspectRatio -> Automatic, PlotLabel -> label,
                          PlotRange -> {{-1.2, 1.2}, {-1.2, 1.2}}];
 Show[GraphicsArray[{
 (* the original square *)
 Graphics[{Thickness[0.001], Line /@ #, Line /@ Transpose[#]}&[
          Map[{Re[#], Im[#]}&, points, {-1}]], opts["z-plane"]],
 (* the mapped square *)
 Graphics[{{GrayLevel[3/4], Disk[{0, 0}, 1]},
          Thickness[0.001], Line /@ #, Line /@ Transpose[#]}&[
          Map[{Re[#], Im[#]}&,
          Map[Function[z, Evaluate[N[%]]], points, {-1}], {-1}]],
         opts["w-plane"]]}]]]]
```

Mathematica has most of the special functions of mathematical physics (see Chapter 3 of the Symbolics volume [1285✱] of the *GuideBooks*) [865✱]. Using elliptic functions, it is possible to find an exact formula for a conformal map from a rectangle to the unit disk [1013✱]. The next graphic visualizes the exact map. To avoid repeating the last input, we modify the last input in a programmatic way and then evaluate the new code.

```
Module[{wExact, k = InverseEllipticNomeQ[Exp[-2. Pi]], K},
        K = EllipticK[k];
       (* the exact conformal map *)
        wExact[z_] := (1 + I JacobiSN[K (z + I), k])/
                      (1 - I JacobiSN[K (z + I), k]);
       (* or another map in elliptic functions:
   {g2, g3} = WeierstrassInvariants[{1., I}];
   wExact[z_] := (1 - I WeierstrassP[(z + (1 + I))/2, {g2, g3}])/
              (1 + I WeierstrassP[(z + (1 + I))/2, {g2, g3}]);
  *)
       (* to obtain above symmetry, apply in addition:
   makeAboveSymmetry[z_] :=
     (3  - Sqrt[2] + I + (2 Sqrt[2] + 1 + (Sqrt[2] + 1) I) z)/
     (2 Sqrt[2] + 1 + (Sqrt[2] + 1) I + (3  - Sqrt[2] + I) z);
  *)
       (* reuse the above input *)
       Last[DownValues[In][[-2]] /.
                (* make changes to last input *)
                HoldPattern[%] -> (* makeAboveSymmetry @ *) wExact[z]]]
```

The typesetting capabilities of *Mathematica* allow mathematical formulas and algorithms to be entered in a still more direct way.

```
ConformalMapSquareToUnitDiskSF[ω_Integer?Positive, z_] :=
Module[{h, H, G, d, l, p, k, t, a, b, ρ,
        C = {-1 + I, 1 + I, 1 - I, -1 - I}},
Ĉ = Partition[Append[C, First[C]], 2, 1];
λ = Total[Abs[Apply[Subtract, Ĉ, {1}]]];
ρ_{j_,k_} = Total[Apply[Abs[#2 - #1] (#1 + t (#2 - #1))^j *
        ((#1 + t (#2 - #1))^k /. c_Complex :→ Conjugate[c]) &, Ĉ, {1}]];
h_{j_,k_} := h_{j,k} = 1/λ ∫_0^1 ρ_{j,k} ⅆt;
H_{n_} := Array[h_{##} &, {n + 1, n + 1}, 0];
d_{n_} := d_n = Det[H_n];
G_{n_}[ξ_] := Array[If[#2 < n, h_{#1,#2}, ξ^#1] &, {n + 1, n + 1}, 0];
l_{n_}[ξ_] := l_n[ξ] = Det[G_n[ξ]];
p_0[ξ_] = 1; p_{n_}[ξ_] := p_n[ξ] = l_n[ξ] / √(d_n d_{n-1});
k[a_, b_] = ∑_{i=0}^ω p_i[a] p_i[b];
Cancel[π/(4 k[0, 0]) Expand[∫_0^z k[0, ξ]^2 ⅆξ]]]
```

`ConformalMapSquareToUnitDiskSF` yields the same result as `ConformalMapSquareToUnitDisk`.

```
ConformalMapSquareToUnitDiskSF[8, z]
```

While the availability of numerical values of special functions is an important part of *Mathematica*, in many instances its problem-solving power arises from connecting numerics, symbolics, and graphics. Here is another simple example: the path of a point vortex in an inviscid fluid in a rectangular region. The path of the vortex $\{x(t), y(t)\}$ is given by the following Hamiltonian system [1305✱], [1327✱] (for spherical rectangles, see [592✱]). Here $\wp(z; g_2, g_3)$ is the Weierstrass $\wp$ function and $g_2(\omega_1, \omega_3)$ and $g_3(\omega_1, \omega_3)$ are the invariants as a function of the half-periods.

$$x'(t) = \frac{\partial H}{\partial y(t)},\; y'(t) = -\frac{\partial H}{\partial x(t)}$$

$$H = -\Gamma \ln(\wp(2\,x(t) + 2\,a;\, g_2(2\,a,\, 2\,i\,b),\, g_3(2\,a,\, 2\,i\,b)) + \wp(2\,y(t) + 2\,b;\, g_2(2\,b,\, 2\,i\,a),\, g_3(2\,b,\, 2\,i\,a)))$$

```
H = -Γ Log[WeierstrassP[2 x[t] + 2 a, {g2, g3}] +
           WeierstrassP[2 y[t] + 2 b, {𝑔2, 𝑔3}]];
```

It is straightforward to get the explicit form of the equations of motions.

```
odes = {x'[t] == D[H, y[t]], y'[t] == - D[H, x[t]]};
odes // TraditionalForm
```

And it is straightforward to solve these equations numerically for different initial conditions. (We choose $\Gamma = 1$, $a = 2$, and $b = 1$ in the following input). The picture shows periodic, self-intersection-free trajectories that are ellipse-shaped for initial conditions near the center and that approximate the rectangle for starting values near the edges.

```
Module[{odesN, T = 3, nsol},
  (* substitute values for Γ, a, and b *)
  odesN = odes /. {{g2, g3} -> WeierstrassInvariants[{2 a, 2 I b}],
                   {𝑔2, 𝑔3} -> WeierstrassInvariants[{2 b, 2 I a}]} /.
                  {a -> 2, b -> 1., Γ -> 1};
Show[(* use different initial conditions of the form {x0, 0} *)
Table[(* solve equations of motion *)
nsol = NDSolve[Join[odesN, {x[0] == x0, y[0] == 0}],
               {x, y}, {t, 0, 2T/x0}, MaxSteps -> 10000];
(* plot the path *)
ParametricPlot[Evaluate[{x[t], y[t]} /. nsol], {t, 0, 2T/x0},
               Axes -> False, DisplayFunction -> Identity,
               PlotStyle -> {{Thickness[0.001], Hue[x0/2.6]}}],
               {x0, 0.1, 1.9, 0.1}],
     DisplayFunction -> $DisplayFunction, Frame -> True,
     FrameTicks -> False, FrameStyle -> {Thickness[0.02]}]]
```

The penultimate example of this subsection deals with a slightly more complicated example: the lines of magnetic induction (which in a 2D cylindrical geometry are also the lines of constant vector potential) of a cylindrical magnet with an air gap. The z component A_z of the vector potential $\mathbf{A}$ (a is the inner radius, b is the outer radius of the magnet, $2\,\pi - 2\,\alpha$ is the slit width, and the slit is pointing into the $-x$ direction) is given by the following sums [1220✱]. We use *Mathematica*'s typesetting capabilities for this example.

```
A[r_, θ_, {a_, b_, α_}] =
 With[{θp = Sin[n α] Cos[n θ] / n^2}, Evaluate //@ Which[
```

$$\begin{array}{ll} \texttt{r > b,} & \alpha\,\mathrm{Log}\left[\frac{b}{b}\right] + \sum_{n=1}^{\infty}\left(\frac{a}{r}\right)^n \theta p - \sum_{n=1}^{\infty}\left(\frac{b}{r}\right)^n \theta p, \\ \texttt{b > r > a,} & \alpha\,\mathrm{Log}\left[\frac{b}{r}\right] + \sum_{n=1}^{\infty}\left(\frac{a}{r}\right)^n \theta p - \sum_{n=1}^{\infty}\left(\frac{r}{b}\right)^n \theta p, \\ \texttt{a > r,} & \alpha\,\mathrm{Log}\left[\frac{b}{a}\right] + \sum_{n=1}^{\infty}\left(\frac{r}{a}\right)^n \theta p - \sum_{n=1}^{\infty}\left(\frac{r}{b}\right)^n \theta p]] \end{array}$$

As the result shows, *Mathematica* was able to sum all three of the above symbolic infinite sums in closed form. The normal component of the field is everywhere differentiable.

```
Plot[Evaluate[A[3 / 2, θ, {1, 2, 7 / 8 π}]],
 {θ, 0, 2 π}, Frame → True, Axes → False,
    PlotStyle → {{GrayLevel[0], Thickness[0.003]}},
 Prolog → {Hue[0], Rectangle[{0, 0.77}, {7 / 8 π, 0.869}],
   Rectangle[{9 π / 8, 0.77}, {2 π, 0.869}]}]
```

The tangential component has a discontinuity in its first derivative at the magnet.

```
Plot[Evaluate[A[r, 0, {1, 2, 7 / 8 π}]], {r, 0, 3}, Frame → True, Axes → False,
    PlotStyle → {{GrayLevel[0], Thickness[0.003]}},
 Prolog → {Hue[0], Rectangle[{1, -0.1}, {2, 2.}]}]
```

The field lines (for a cylindrical geometry, they are the equi-A_z-potential lines) are shown in the following graphics. The homogeneous field in the air gap is nicely visible (although running the following input will take a few minutes).

```
ContourPlot[Evaluate[Re[A[√(x² + y²), ArcTan[x, y], {1, 2, 7 / 8 π}]]],
    {x, -3, 3}, {y, -3, 3}, PlotPoints → 160,
    Contours → Range[-0.1, 2.2, 0.1], ContourShading → False,
    Compiled → False, FrameTicks → None,
 ContourStyle → {{Thickness[0.001], GrayLevel[0]}},
    Prolog → {Thickness[0.006], Hue[0], Disk[{0, 0}, 2, {- 7 / 8 π, 7 / 8 π}],
                  GrayLevel[1], Disk[{0, 0}, 1, {-7 π, 7 π} / 8]}]
```

Here is a 3D picture of the field strength.

```
ListPlot3D[(* take out data from last graphic *) First[%],
 PlotRange → All, Mesh → False, ViewPoint → {-2, -2, 2}, Axes → False]
```

We continue with another, slightly larger application from electrodynamics. Localized [829✱], propagating solutions of the free Maxwell equations can be derived from the simple vector potential **A** = curl {0, 0, *ψ*(*x*, *y*, *z*; *t*)} where the scalar function *ψ*(*x*, *y*, *z*; *t*) is a simple rational function of *x*, *y*, *z*, and *t* [831✱]. Here is such a function; *a* and *b* are parameters determining the shape of the field, and *ψ0* is a normalization constant.

```
ψZiolkowski[{x_, y_, z_}, t_] =
            a b ψ0/(x^2 + y^2 + (a - I (z + c t)) (b + I (z - c t)));

ΨLekner[{x_, y_, z_}, t_] =
        (x + I y)/(b + I (z - c t)) ψZiolkowski[{x, y, z}, t]
```

Using $\mathbf{E} = -\partial / \mathbf{A}\, \partial t$ and $\mathbf{B} = \text{curl}\, \mathbf{A}$, we can derive the following (complex-valued) electric and magnetic fields.

```
ℰℂ[{x_, y_, z_}, t_] = {-1/c D[ΨLekner[{x, y, z}, t], y, t],
                          1/c D[ΨLekner[{x, y, z}, t], x, t], 0} //
                                                           Together;

ℬℂ[{x_, y_, z_}, t_] =
 { D[ΨLekner[{x, y, z}, t], x, z],  D[ΨLekner[{x, y, z}, t], y, z],
  -D[ΨLekner[{x, y, z}, t], x, x] - D[ΨLekner[{x, y, z}, t], y, y]} //
                                                               Together;
```

Here is a quick check of the Maxwell equations themselves using the just-derived fields.

```
(* the curl vector analysis operation *)
curl[{a_, b_, c_}, {x_, y_, z_}] :=
     {D[c, y] - D[b, z], D[a, z] - D[c, x], D[b, x] - D[a, y]}

(* the div vector analysis operation *)
div[{a_, b_, c_}, {x_, y_, z_}] := D[a, x] + D[b, y] + D[c, z]
```

```
With[{ℰ = ℰℂ[{x, y, z}, t], ℬ = ℬℂ[{x, y, z}, t]},
     (* the four free-space Maxwell equation *)
     {div[ℰ, {x, y, z}], div[ℬ, {x, y, z}],
      curl[ℰ, {x, y, z}] + 1/c D[ℬ, t],
      curl[ℬ, {x, y, z}] - 1/c D[ℰ, t]} // Together]
```

We form real-valued fields by taking the real (or imaginary) part of ℰℂ and ℬℂ.

```
{ℰ[{x_, y_, z_}, t_], ℬ[{x_, y_, z_}, t_]} = Function[ℰℂℬv,
    ComplexExpand[Re[ℰℂℬv[{x, y, z}, t]], TargetFunctions -> {Re, Im}] //
                                         Together // Factor] /@ {ℰℂ, ℬℂ};
```

Although still rational functions in x, y, z, and t, the real-valued fields are quite large expressions. The following measures the number of independent subexpressions present in ℰ and ℬ.

```
{LeafCount[ℰ[{x, y, z}, t]], LeafCount[ℬ[{x, y, z}, t]]}
```

By expressing the fields in cylindrical coordinates, and simplifying the results, we obtain considerably smaller expressions.

```
(* change to polar coordinates and simplify resulting expressions;
   using some more specific functions the simplification could be
   made faster *)
{LeafCount[ℰc[{r_, φ_, z_}, t_] = Simplify[ℰ[{r Cos[φ], r Sin[φ], z}, t]]],
 LeafCount[ℬc[{r_, φ_, z_}, t_] = Simplify[ℬ[{r Cos[φ], r Sin[φ], z}, t]]]}
```

The function `𝓊ct` is the time-dependent energy density and of the localized field configuration and `𝓅zt` is the time-dependent z-component of the momentum density of the field. For the following calculations and visualizations, we specialize the parameters to $a = 2$, $b = 1$ and assume that the speed of light c is 1.

```
fieldParameterRules =  {ψ0 -> 1, c -> 1, a -> 2, b -> 1};

𝓊ct[{r_, φ_, z_}, t_] = 1/(8 Pi) (ℰc[{r, φ, z}, t].ℰc[{r, φ, z}, t] +
                                 ℬc[{r, φ, z}, t].ℬc[{r, φ, z}, t]) /.
                                                fieldParameterRules;

𝓅zct[{r_, φ_, z_}, t_]  =
    1/(4 Pi c) Cross[ℰc[{r, φ, z}, t], ℬc[{r, φ, z}, t]][[3]] /.
                                                fieldParameterRules;
```

We obtain the total energy U and the total momentum P_z by integrating the energy and momentum densities. (Because they are both conserved quantities, we can choose $t = 0$ [830✱].)

```
(* energy density at t = 0 *)
𝓊c0 = Simplify[Together[𝓊ct[{r, φ, z}, 0]]];

(* z-component of the momentum density at t = 0 *)
𝓅zc0 = Simplify[Together[𝓅zct[{r, φ, z}, 0]]];

(* total energy *)
Integrate[r 𝓊c0, {z, -Infinity, Infinity}, {r, 0, Infinity}, {φ, 0, 2Pi},
          GenerateConditions -> False]

(* total momentum *)
Integrate[r 𝓅zc0, {z , -Infinity, Infinity}, {r, 0, Infinity}, {φ, 0, 2Pi},
          GenerateConditions -> False]
```

Interestingly, we have $U > c\ P_z$ (this means this electromagnetic packet cannot be interpreted as a photon). The next graphics show lines of constant energy density in the x,y-plane and in the x,z-plane. The colors indicate increasing time, from red to blue. We see the propagation along the z-axis and the overall spreading of the field packet and a twist around the z-axis.

```
Show[GraphicsArray[
Block[{$DisplayFunction = Identity,
          (* common options for the next two plots *)
          copts = Sequence[PlotPoints -> 200, Contours -> 10,
                           ContourShading -> False, ContourStyle -> Hue[t/4]]}
  {(* constant energy density in the x-y-plane *)
   Show[Table[ContourPlot[Evaluate[𝓊ct[{Sqrt[x^2 + y^2], ArcTan[x, y], 0}, t
             {x, -5, 5}, {y, -5, 5}, Evaluate[copts]], {t, 0, 3}]],
   (* constant energy density in the x-z-plane *)
   Show[Table[ContourPlot[Evaluate[𝓊ct[{Sqrt[x^2 + 0^2], ArcTan[x, 0], z}, t
            {x, -5, 5}, {z, -5, 5}, Evaluate[copts]], {t, 0, 3}]]}]]]
```

The typesetting capabilities of *Mathematica* allow to create new notations and to use them in programming. Here is a simple example from quantum mechanics. For implementing more complicated notations, the notations package comes handy. In general, in the *GuideBooks,* we will not resort to typeset input to guarantee a 1–1 correspondence between the format of the (printed) inputs and their meaning. We implement abstract quantum mechanical state vectors (kets [388✱]) as `|i⟩`$_A$ `= Ket[A, i]` (the first letter `A` labels the particle and `i` its quantum state).

```
(∗ do not numericalize inside kets ∗)
SetAttributes[Ket, NHoldAll]

(∗ accept |ψ⟩_A as input ∗)
MakeExpression[
  SubscriptBox[RowBox[{RowBox[{"|", ψ_}], "⟩"}], A_], form_] :=
 MakeExpression[RowBox[{"Ket", "[", A, ",", ψ, "]"}], form]

(∗ format Ket[A, ψ] in output as |ψ⟩_A ∗)
MakeBoxes[Ket[A_, ψ_], form_] :=
 StyleBox[
  SubscriptBox[RowBox[{RowBox[{"|", MakeBoxes[ψ, form]}], "⟩"}], A],
          AutoStyleOptions → {"UnmatchedBracketStyle" → None}]
```

`|ψ⟩`$_{AB}$ is a nonseparable two-particle state from the tensor product of two four-dimensional spaces.

$$|\psi\rangle_{AB} = \sum_{i=1}^{4}\sum_{j=1}^{4} \mathrm{Cos}\left[\frac{i}{j} + \frac{j}{i}\right] |i\rangle_A\, |j\rangle_B$$

The following short program writes a given two-particle state (in general form $\sum_{i,j=1}^{d} c_{i\,j}\, |i\rangle_A\, |j\rangle_B$) in Schmidt form $\sum_{j=1}^{d} c_j\, |j\rangle_A\, |j\rangle_B$ (see [510✱], [1429✱], [1185✱], [207✱], [1055✱], [1328✱], [857✱], [426✱] for details and [1427✱], [1428✱], [1182✱] for envariance applications). The function `SchmidtDecomposition` returns the Schmidt form of the input state and how the new vectors $|j\rangle_A$ and $|j\rangle_B$ are expressed through the original vectors. (A singular value decomposition is at the heart of the function `SchmidtDecomposition`.) Inside the program, we use the above-defined `|i⟩`$_A$.

```
SchmidtDecomposition[ψ_, {u_, v_}] :=
 Module[{allKets, AKets, BKets, A, B, allKetΠs, M, U, Ω, V, d},
  (* kets occurring in the original state vector *)
  allKets = Union[Cases[ψ, _Ket, ∞]];
  (* kets of the two subsystems *)
  {AKets, BKets} = Split[allKets, #1[[1]] === #2[[1]] &];
  (* subsystem labels *)
  {A, B} = {AKets[[1, 1]], BKets[[1, 1]]};
  (* coefficient matrix *)
  allKetΠs = Outer[List, AKets, BKets];
  M = Map[ψ /. Thread[# -> {1, 1}] /. _Ket -> 0 &, allKetΠs, {2}];
  (* singular value decomposition of coefficient matrix *)
  {U, Ω, V} = SingularValues[N[M, $MachinePrecision + 1]];
  {(* Schmidt decomposed vector *) Sum[Ω[[j]] |u_j〉_A |v_j〉_B, {j, 1, Length[Ω]}],
   (* new basis vectors expressed through old vectors *)
   Join[MapIndexed[|u_#2[[1]]〉_A -> #.AKets &, U],
        MapIndexed[|v_#2[[1]]〉_B -> #.BKets &, V]]}]
```

Here is the Schmidt form of the above state $|\psi\rangle_{AB}$.

```
|ψ〉_AB // SchmidtDecomposition[#, {u, v}] & // (sd = #) & // N //
 TraditionalForm
```

Expressing the new basis vectors $|u_j\rangle_A$ and $|v_j\rangle_B$ through the old ones allows for a quick check of the decompositions.

```
|ψ〉_AB - ReplaceAll @@ sd // Expand
```

We use the function `SchmidtDecomposition` for one more calculation: The Schmidt coefficients of a state $\sum_{i=1}^{d}\sum_{j=1}^{d} c_{i,j}\,|i\rangle_A\,|j\rangle_B$ where the $c_{i,j}$ are random coefficients with normal distributions are in average slowly decreasing functions of the index. The next graphic shows the Schmidt coefficients for 100 initial two-particle states and $d = 12$.

```
schmidtCoefficients =
  Table[ψ = Sum[InverseErf[Random[Real, {-1, 1}]] |i〉_A |j〉_B, {i, 1, 12}, {j, 1, 12}];
        DeleteCases[List @@ SchmidtDecomposition[ψ, {u, v}][[1]], |_〉_ , ∞],
   {100}];

Show[Graphics[MapIndexed[Point[{#2[[2]], #1}] &, schmidtCoefficients, {2}]],
     PlotRange -> All, Frame -> True, Axes -> False]
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

■ 1.2.4 Programming

In addition to numeric and symbolic computations and generating graphics, *Mathematica* provides a general programming and development environment. Large (several pages or screens), and even very large, programs can be written in *Mathematica*, although these programs typically will be *much* shorter than they would be in other programming languages. Such programs may involve all of the capabilities of *Mathematica*, including numerical and symbolic calculations, pattern matching, graphics, variable name protection, etc. Two examples of larger programs from physics are FeynCalc (for doing high-energy physics calculations), http://www.mertig.com and MathTensor (for doing general relativity calculations), http://smc.vnet.net/MathTensor.html. Also, all of the *Mathematica* Application Library packages, http://store.wolfram.com/catalog/apps, are written in *Mathematica*.

Let us start with a (very) small program. Given a real number x with $0 < x < 1$, we want to extract the l first digits in base b. The following code gives a recursive definition for extracting the digits.

```
realDigits[x_ /; 0 < x < 1, base_, l_] :=
Module[{rest, digit},
          (* recursion initial *)
          rest[1] = FractionalPart[x];
          (* recursion for remaining part and digit *)
          rest[n_] := rest[n] = FractionalPart[rest[n - 1] base];
          digit[n_] := digit[n] = Floor[rest[n] base];
          (* list of digits *) Table[digit[i], {i, l}]];
```

Here is a simple test for the above program.

```
realDigits[N[Pi - 3, 200], 10, 100]
```

Mathematica also has a built-in function for getting the digits of a real number. It returns the same result.

```
RealDigits[N[Pi - 3, 200], 10, 100][[1]]
```

Next, we carry out a highly recursive calculation. The number of ways $p_m(n)$ to decompose a positive integer n into m positive integers k_j, such that $n = \sum_{j=1}^{m} k_j$ obeys the recursion $p_m(n) = \sum_{k=1}^{\min(n-m,m)} p_{n-m}(k)$ [36✱], [1362✱]. The function `pList` returns a list of the nonvanishing $p_m(n)$ for a given n.

```
pList[n_Integer?Positive] :=
Block[{(* for larger n *) $RecursionLimit = Infinity, p},
       p[ν_, ν_] := p[ν, ν] = 1; (* the case n = 1 + 1 + ... + 1 *)
    (* remember intermediate values of p *)
       p[ν_, μ_] := p[ν, μ] = Sum[p[ν - μ, k], {k, Min[ν - μ, μ]}];
       (* all nonzero values for 1 ≤ m ≤ n *) Table[p[n, μ], {μ, n}]]
```

Here are the values of $p_1(10)$, $p_2(10)$, …, $p_{10}(10)$.

```
pList[10]
```

Calculating all nonzero values of $p_m(1000)$ takes a few minutes and requires the calculation of more than 250000 intermediate values of the $p_m(n)$. The next graphic shows $p_m(1000)$ as a function of m.

```
ListPlot[pList[1000], PlotRange -> {{0, 400}, All}]
```

Every introductory chapter on *Mathematica* should include a definition of the factorial function $n \longrightarrow n!$, $n \in \mathbb{N}$. The obvious one `f[n_] := f[n] = n f[n -1]` with the initial condition `f[0] = 1` is short, but suffers from a nonoptimal complexity for larger n [175✱]. The following, slightly more complicated definition, is very efficient. It is based on extracting all powers of 2 and carrying out the multiplication of the remaining odd numbers by binary splitting [599✱].

```
factorial[n_] := 2^(n - DigitCount[n, 2, 1])*
                 Product[p[n/2^k, n/2^(k - 1)]^k, {k, 1, Floor[Log[2, n]]}]

(* form recursively product of odd numbers between m and n *)
p[m_, n_] := p[m, Round[(m + n)/2]] p[Round[(m + n)/2], n] /;  n - m > 5

p[m_, n_] := Product[2j + 1, {j, Ceiling[Floor[m]/2], Floor[(n - 1)/2]}]
```

The built-in factorial function `Factorial` is, of course, faster than `factorial`, but for $n = 10^6$, the difference is only a factor of two.

```
{N[Timing[Factorial[1000000]]], N[Timing[factorial[1000000]]]}
```

Here is another small programming example. The Bolyai expansion of a real number x is a nested root of the form [1128✱], [906✱]

$$x = a_0 - 1 + \sqrt[m]{a_1 + \sqrt[m]{a_2 + \sqrt[m]{a_3 + \cdots}}} \;.$$

The Bolyai digits a_k are integers $0 \leq a_k \leq 2^m - 1$. The following concise input calculates the Bolyai expansion of x using n roots of order m.

```
BolyaiRoot[x_?((NumericQ[#] && Precision[#] === Infinity &&
                Not[IntegerQ[x]])&),
           m_Integer?Positive, n_Integer?Positive] :=
Block[{$MaxExtraPrecision = 1000},
IntegerPart[x] - 1 + Fold[(#1 + #2)^(1/m)&, 0,
        Reverse[IntegerPart[(1 + #)^m - 1]& /@
             NestList[FractionalPart[(1 + #)^m - 1]&,
                                FractionalPart[x], n]]]]
```

Here are three examples: The outermost ten roots for π for $m = 2$, $m = 10$, and $m = 99$.

```
b2  = BolyaiRoot[Pi, 2, 10]

b10 = BolyaiRoot[Pi, 10, 10]

b99 = BolyaiRoot[Pi, 99, 10]
```

The difference between the nested roots and π is a decreasing function of m.

```
Block[{$MaxExtraPrecision = 1000}, N[{b2, b10, b99} - Pi, 22] // N]
```

Here is a straightforward definition of a Pfaffian [223✱], [874✱], [360✱], [770✱], [600✱] (the ↑ under the element a_j indicates that this element is removed).

$$\operatorname{Pf}(a_1, a_2, \ldots, a_{2n}) = \sum_{k=2}^{2n} \operatorname{Pf}(a_1, a_j)\, \operatorname{Pf}\left(a_2, \ldots, \underset{\uparrow}{a_j}, \ldots, a_{2n}\right)$$

$$\operatorname{Pf}(a_1, a_2) = -\operatorname{Pf}(a_2, a_1)$$

```
Pf[as:{a1_, a2_, __}] :=
   Sum[(-1)^j Pf[{a1, as[[j]]}] Pf[Delete[as, {{1}, {j}}]],
       {j, 2, Length[as]}] /; EvenQ[Length[as]]

Pf[{x_, y_}] := -Pf[{y, x}] /; Not[OrderedQ[{x, y}]]
```

The next input calculates the explicit expanded Pfaffian of $\operatorname{Pf}(a_1, a_2, \ldots, a_6)$.

```
Pf[Table[Subscript[a, j], {j, 6}]] // Expand
```

```
determinantThroughPfaffian[m_?(MatrixQ[#] && Equal @@ Dimensions[m]&)] :=
With[{d = Length[m]},
 Pf[Join[Array[1, d], Reverse @ Array[2, d]]] /.
                     Pf[{1[j_], 2[k_]}] :> m[[j, k]] /. _Pf -> 0]
```

The determinant of a matrix $\mathbf{A} = (a_{i,j})_{1\leq i,j\leq n}$ can be expressed through the Pfaffian in the following form:

$$\det(\mathbf{A}) = \mathrm{Pf}\big(b_1, b_2, \ldots, b_{n-1}, b_n, \tilde{b}_n, \tilde{b}_{n-1}, \ldots, \tilde{b}_2, \tilde{b}_1\big)$$

and the rules $\mathrm{Pf}(b_i, b_j) = \mathrm{Pf}(\tilde{b}_i, \tilde{b}_j) = 0$, $\mathrm{Pf}(b_i, \tilde{b}_j) = a_{i,j}$.

Here is a symbolic 6×6 matrix.

```
(A = Table[Subscript[a, i, j], {i, 6}, {j, 6}]) // TableForm
```

The next input checks the above determinant formula.

```
determinantThroughPfaffian[A] - Det[A] // Expand
```

Mathematica is frequently also an ideal tool to prototype and analyze algorithms. Here we will give a simple sorting algorithm. The so-called bead-sort algorithm orders a list of k positive integers increasingly [50✱], [51✱]. An integer n is initially represented as a list of n 1's, each 1 standing for a bead. The k initial integers to be sorted are in the beginning represented as left-aligned rows of beads. In each step of the sorting process, a bead slides down one unit if possible (like in an 90°-rotated abacus) until each bead can no longer slide. The following function `beadSortStep` implements one step of the bead-sort algorithm. Using functional programming constructs, we can deal with rows of beads at once instead of explicitly looping over the rows and columns of beads.

```
(* the argument of beadSortStep is a rectangular array of
  0's and 1's; the ones are the beads *)
beadSortStep = With[{l = Length[First[#]]}, Transpose[Map[
(* bead slides down if possible *)
If[MatchQ[#, {0, 0, 0} | {0, 0, 1} | {0, 1, 0} | {1, 1, 0}], 0, 1]&,
   (* rows and lower and upper neighbor rows *) Partition[#, 3, 1]& /@
   Transpose[Join[{Table[0, {l}]}, #, {Table[1, {l}]}]], {2}]]]&;
```

The function `toBeads` converts a list of integers into rows of beads (0 indicates the absence of a bead). The function `fromBeads` converts from the beads to integers.

```
(* convert list of integers to lists of beads *)
toBeads[l_] := Join[Table[1, {#}], Table[0, {Max[l] - #}]]& /@ l
(* convert lists of beads to list of integers *)
fromBeads[l_] := Count[#, 1]& /@ l
```

To visualize the bead-sort steps, we define a function `beadGraphics`.

```
beadGraphics[beads_] := Graphics[
{(* rods on which the beads slide *)
 {GrayLevel[1/2], Table[Line[{{k, -1}, {k, -Length[beads] - 1/2}}],
                        {k, Length[beads[[1]]]}]},
 (* the beads *)
 MapIndexed[If[#1 === 1, Disk[Reverse[#2 {-1, 1}], 0.4], {}]&,
              beads, {2}]},
              AspectRatio -> Automatic, PlotRange -> All]
```

The next five graphics show how the bead-sort algorithm orders the list {7, 2, 1, 4, 2}. The function `FixedPoint⋱ List` applies the step `beadSortStep` until the beads are sorted.

```
(* the steps of the sorting process *)
sortHistory = Drop[FixedPointList[beadSortStep,
                                  toBeads[{7, 2, 1, 4, 2}]], -1];
(* display the steps *)
Show[GraphicsArray[beadGraphics /@ sortHistory]]
```

In intermediate step, we can have rows exhibiting a number of beads not in the initial list of integers.

```
Map[fromBeads, sortHistory]
```

Using the just-implemented functions, we define a function `BeadSort` that sorts a list of nonnegative integers.

```
BeadSort[l_?(VectorQ[#, (IntegerQ[#] ∧ NonNegative[#])&]&)] :=
        fromBeads[FixedPoint[beadSortStep, toBeads[l]]]
```

Here is an example of `BeadSort` in action.

```
BeadSort[{12, 6, 1, 8, 3, 2, 7, 1, 5, 0, 2}]
```

The next input is an example of a slightly larger program. This is the typical appearance of a larger piece of *Mathematica* source code. In essence, it consists of the following parts:

- Explanation of how to use it
- Commands to load needed packages
- Definition of auxiliary functions
- Definition of the actual (exportable) functions with a check for the appropriateness of its arguments
- Implementation of warnings for inappropriate variables, error messages, etc.

Such a program will usually have context declarations at the beginning and the end to provide protection for the local variables, and will be deposited in the directory of the user's packages (we discuss these issues in Chapter 4). Assuming it has been placed in a special directory by the user, it can be loaded using the function `Needs`, as in `Needs["`*directory*`` `ChainedPlatonicBody`"] ``.

```
(* Information on the functions implemented below
   can be obtained with ?InPlaneTori and ?NormalPlaneTori *)

InPlaneTori::usage =
"InPlaneTori[platonicSolid, φ10:0, φ20:0, r1rel:0.68, r2rel:0.12, n1:Automa
 \n \n φ10 and φ20 vary in the ranges: -2 Pi/n1 <= φ10 <= 2 Pi/n1.";

NormalPlaneTori::usage =
"NormalPlaneTori[platonicSolid, φ10:0, φ20:0, r1rel:0.68, r2rel:0.12, n1:Au

(* Read in the necessary package *)
Needs["Graphics`Polyhedra`"]

(* Turn off the warnings *)
Off[General::spell]; Off[General::spell1];

(* Cancel other function definitions with the same names *)
Clear[center, normalize, faces, uniList, toPolygons, neighborList,
      InPlaneTori, NormalPlaneTori];

(* Definition of auxiliary functions and the two functions
   InPlaneTori and NormalPlaneTori to be "exported" *)

(* center of gravity of a face *)
center /:
```

```
center[face_List] := Mean[face];

(* normalize a vector to unit length *)
normalize /:
normalize[vector_?(VectorQ[#, NumericQ]&)] := N[vector]/Norm[vector];

(* faces of a Platonic solid *)
faces /:
faces[platonicSolid: (Cube | Tetrahedron | Octahedron |
                      Dodecahedron | Icosahedron)] :=
faces[platonicSolid] =
If[With[{mp = Mean[#]}, Cross[#[[1]], #[[2]]].mp] < 0,
   #, Reverse[#]]& /@ (First /@ N[First[Polyhedron[platonicSolid]]]);

(* make a single torus *)
uniList /:
uniList[φ10_, n1_, r1_, φ20_, n2_, r2_] :=
Module[{cφ1tab, sφ1tab, cφ2tab, sφ2tab, auxx, auxy, auxz, pi = N[Pi]},
         (* calculate points *)
         {cφ1tab, sφ1tab, cφ2tab, sφ2tab} =
         Table[N @ #1[φ], {φ, #2, #2 + 2pi (1 - 1/#3), 2pi/#3}]& @@@
           {{Cos, φ10, n1}, {Sin, φ10, n1}, {Cos, φ20, n2}, {Sin, φ20, n2}};
         (* form polygons from points *)
         auxx = r1 Transpose[Table[cφ1tab, {n2}]] +
                r2 Outer[Times, cφ1tab, cφ2tab];
         auxy = r1 Transpose[Table[sφ1tab, {n2}]] +
                r2 Outer[Times, sφ1tab, cφ2tab];
         auxz = r2 N[Cos[pi/n1]] Table[sφ2tab, {n1}];
         MapThread[List, {auxx, auxy, auxz}, 2]];

(* make polygons from list of points *)
toPolygons /: toPolygons[points:(p0_List)] :=
Module[{p1 = RotateLeft /@ p0, p2 = RotateLeft[p0], p3},
          p3 = RotateLeft /@ p2;
          Flatten[MapThread[Polygon[{#1, #2, #3, #4}]&,
                            {p0, p1, p3, p2}, 2]]];

(* the tori in the planes of the faces *)
InPlaneTori /:
InPlaneTori[platonicSolid:(Cube | Tetrahedron | Octahedron |
                           Dodecahedron | Icosahedron),
            φ10_:0, φ20_:0, r1rel_:0.68, r2rel_:0.12,
            n1_:Automatic, n2_:Automatic] :=
 Module[{allFaces, oneFace, cen, dis, λ, num1, num2,
         dirx, diry, dirz, uni, polys},
         (* data of the Platonic solid *)
         allFaces = faces[platonicSolid]; oneFace = allFaces[[1]];
         cen = center[oneFace]; λ =  Length[oneFace];
         {num1, num2} = If[# === Automatic, λ, #]& /@ {n1, n2};
         l = Sqrt[(oneFace[[1]] - cen).(oneFace[[1]] - cen)];
         r1 = l r1rel; r2 = l r2rel;
         (* make tori *)
         uni = uniList[φ10, num1, r1, φ20, num2, r2];
         polys = toPolygons[uni];
         Table[oneFace = allFaces[[i]];
               cen = center[oneFace];
               (* three orthogonal directions *)
              {dirx, diry} = normalize[oneFace[[#]] - cen]& /@ {1, 2};
               diry = normalize[diry - dirx (diry.dirx)];
```

```
             dirz = normalize[cen];
             Map[(cen + #.{dirx, diry, dirz})&,
                  polys, {3}], {i, Length[allFaces]}]] /;
       (* test arguments *)
       (NumberQ[N[φ10]]   && NumberQ[N[φ20]] &&
        NumberQ[N[r1rel]] && NumberQ[N[r2rel]] &&
        ((IntegerQ[n1] && n1 > 2) || n1 === Automatic) &&
        ((IntegerQ[n2] && n2 > 2) || n2 === Automatic));

(* neighboring faces *)
neighborList /:
neighborList[platonicSolid:(Cube | Tetrahedron | Octahedron |
                            Dodecahedron | Icosahedron)] :=
Module[{fc, allPairs, allPairTypes, where},
       (* data specific to the Platonic solid *)
       fc = faces[platonicSolid];
       allPairs = Table[Flatten[
        Table[{fc[[k]][[i]], fc[[k]][[j]]},
              {i, Length[fc[[k]]]}, {j, i - 1}], 1], {k, Length[fc]}];
       allPairTypes = Map[Sort, allPairs, {2}];
       where = Map[Position[allPairTypes, #]&, allPairTypes, {2}];
       where = Select[Flatten[where, 1], (Length[#] != 1)&];
       Union[Map[First[Transpose[#]]&, where]]];

(* the tori in the planes perpendicular to the faces *)
NormalPlaneTori /:
NormalPlaneTori[platonicSolid:(Cube | Tetrahedron | Octahedron |
                               Dodecahedron | Icosahedron),
                φ10_:0, φ20_:0, r1rel_:0.68, r2rel_:0.12,
                n1_:Automatic, n2_:Automatic] :=
 Module[{allFaces, nl, fa, fa1, fa2, λ, vert, cen1, cen2, l,
         dirx, diry, dirz, num1, num2, polys, uni},
       (* data specific to the Platonic solid *)
       allFaces = faces[platonicSolid];
       fa = Faces[platonicSolid];
       vert = N[Vertices[platonicSolid]];
       nl = neighborList[platonicSolid];
       fa1 = allFaces[[1]]; λ = Length[fa1];
       {num1, num2} = If[# === Automatic, λ, #]& /@ {n1, n2};
       cen1 = center[fa1];
       l = Sqrt[(fa1[[1]] - cen1).(fa1[[1]] - cen1)];
       r1 = l r1rel; r2 = l r2rel;
       (* make tori *)
       uni = uniList[φ10, num1, r1, φ20, num2, r2];
       polys = toPolygons[uni];
       Table[{fa1, fa2} = allFaces[[nl[[i, {1, 2}]]]];
             {cen1, cen2} = {center[fa1], center[fa2]};
             aux = Intersection[fa[[nl[[i, 1]]]], fa[[nl[[i, 2]]]]];
             mp = (vert[[aux[[1]]]] + vert[[aux[[2]]]])/2;
             (* three orthogonal directions *)
             dirx = normalize[mp]; diry = mp - cen1;
             diry = normalize[diry - dirx (diry.dirx)];
             dirz = normalize[vert[[aux[[1]]]] - mp];
             Map[(mp + #.{dirx, diry, dirz})&, polys, {3}],
             {i, Length[nl]}]] /; (* test arguments *)
              (NumberQ[N[φ10]]   && NumberQ[N[φ20]] &&
               NumberQ[N[r1rel]] && NumberQ[N[r2rel]] &&
              ((IntegerQ[n1] && n1 > 2) || n1 === Automatic) &&
              ((IntegerQ[n2] && n2 > 2) || n2 === Automatic))
```

We can get the syntax for the two functions `InPlaneTori` and `NormalPlaneTori` defined above by typing a `?` before the function name.

```
?InPlaneTori
```

```
?NormalPlaneTori
```

The program is fast, taking only a few seconds, but the graphical display may take a bit longer, depending on the computer used. Here is the measured time for the computation of 20 triangular tori on the faces of an icosahedron.

```
Timing[ico = InPlaneTori[Icosahedron];]
```

We now give a few examples using this program. The functions `InPlaneTori` and `NormalPlaneTori` only produce a list of polygons; they do not generate a graphic.

```
Short[ico // OutputForm, 10]
```

These polygons still have to be displayed using `Show[Graphics3D[...],` *optionsForThePlot*`]`. We look at `ico`, together with a few other examples.

```
Show[GraphicsArray[{
 (* the icosahedron *)
 Graphics3D[ico, Boxed -> False],
 (* the cube *)
 Graphics3D[{InPlaneTori[Cube, 0, Pi/4, 0.65, 0.17],
             NormalPlaneTori[Cube, 0, Pi/4, 0.65, 0.17]},
            Boxed -> False],
 (* the octahedron *)
 Graphics3D[{Hue[Random[]], #}& /@ (* add color *)
             InPlaneTori[Octahedron, 0, 0, 0.5, 0.2, 3, 4],
            Lighting -> False, Boxed -> False],
 (* another icosahedron *)
 Graphics3D[{InPlaneTori[Icosahedron, 0, 0, 0.58, 0.1],
             NormalPlaneTori[Icosahedron, Pi/3, 0, 0.58, 0.1]},
            Boxed -> False]}, GraphicsSpacing -> -0.12]]
```

The functions `InPlaneTori` and `NormalPlaneTori` compute the polygons of the tori to be plotted. *Mathematica* offers numerous of possibilities to determine the appearance (e.g., color, form of the edges, etc.).

```
Show[Graphics3D[{EdgeForm[{Thickness[0.001], Hue[0.7]}],
                 SurfaceColor[Hue[0.2], Hue[0.1], 2],
                 {InPlaneTori[Dodecahedron, 0, 0, 0.64, 0.1],
                  NormalPlaneTori[Dodecahedron, 0, 0, 0.64, 0.1]}}],
      Boxed -> False, Prolog -> {GrayLevel[0], Disk[{1/2, 1/2}, 0.34]}]
```

Once one has a graphic with one (or more) continuously changeable parameter, it is straightforward to generate an animation. We can change the orientation of the tori. In addition, we will add some coloring.

```
SeedRandom[7777777];
colors[φ_] = (* φ-dependent colors *)
Table[{EdgeForm[{Thickness[0.001], Hue[# + 1/2]}],
       SurfaceColor[Hue[#], Hue[Random[] + Random[]/3 Sin[φ]],
                    3 Random[]]}&[Random[] + Random[]/3 Sin[φ]], {42}];
```

```
rotatingToriGraphics[φ_] := Graphics3D[Flatten /@
 (* add color to each torus *)
 Transpose[{colors[φ], Join[InPlaneTori[Dodecahedron, φ, φ, 0.64, 0.11],
                           NormalPlaneTori[Dodecahedron, φ, φ, 0.64, 0.11]]}],
  ViewPoint -> {2Cos[φ], 2Sin[φ], 1.5}, Background -> GrayLevel[0.8],
  Boxed -> False, SphericalRegion -> True,
  PlotRange -> 1.5{{-1, 1}, {-1, 1}, {-1, 1}}]
```

```
With[{frames = 6},
Show[GraphicsArray[#]]& /@ Partition[Table[rotatingToriGraphics[φ],
                            {φ, 0, 2Pi (1 - 1/frames), 2Pi/frames}], 3]]
```

```
With[{frames = 120}, Do[Show[rotatingToriGraphics[φ]],
                        {φ, 0, 2Pi (1 - 1/frames), 2Pi/frames}]];
```

Let us implement another animation example. This time the implementation will be smaller, but the computational effort per graphic will be considerably larger. We will visualize the equipotential surfaces of a charged icosahedral wireframe. We normalize the potential in such a way that the potential φ at center has the value

$$\varphi(\{0, 0, 0\}) = \varphi^* = 15\left(\left(5+\sqrt{5}\right)/2\right)^{1/2} \ln\!\left(2\left(5-2\sqrt{5}\right)^{1/2} - \sqrt{5} + 4\right) \approx 33.33798\ldots \approx 100/3.$$

(In our units, this corresponds to a unit charge of an icosahedron whose vertices have unit distances to the origin.) By visualizing the surfaces $\varphi(\{x, y, z\}) = c$ as a function of the parameter c, we obtain an animation.

The following inputs generate a compiled function that can quickly calculate the potential $\varphi(\{x, y, z\})$.

```
Needs["Graphics`Polyhedra`"]

(* rotate and rescale standard icosahedron *)
γ = -ArcCos[Sqrt[1/3 + 2/(3 Sqrt[5.])]];
ℛ = {{Sin[γ], 0, Cos[γ]}, {-Cos[γ], 0, Sin[γ]}, {0, -1, 0}} // N;
ℛInv = Inverse[ℛ];
ico = Map[ℛ.(#/Sqrt[#.#])&, Polyhedron[Icosahedron][[1]], {-2}];
(* edges of the rescaled icosahedron *)
edges = Union[Sort /@ Flatten[Partition[#[[1]], 2, 1]& /@ ico, 1]];

(* potential of a line segment *)
potentialφ[{𝓍0:{x0_, y0_, z0_}, 𝓍1:{x1_, y1_, z1_}}, 𝓍:{x_, y_, z_}] =
With[{a = #.#&[𝓍0 - 𝓍1], b = 2(𝓍 - 𝓍0).(𝓍0 - 𝓍1), c = #.#&[𝓍 - 𝓍0]},
     (Log[(2a + b + 2Sqrt[a(a + b + c)])/(b + 2Sqrt[a c])])/Sqrt[a]];

(* compiled form of the potential *)
icoφC = Compile[{x, y, z}, Evaluate[
                 Plus @@ (potentialφ[#, {x, y, z}]& /@ edges)]];
```

Because of the symmetry of an icosahedron, we will calculate the 1/120th part of the equipotential surface directly and will generate the remaining parts using rotations. The following functions implement the corresponding change of variables to a coordinate system adapted to cover a 1/120th of the full solid angle.

```
(* definitions for symbols φ𝕞, 𝕪𝕫, ℙ1, 𝕕, 𝕗, and toXYZ *)
Module[{xm, ym, zm},
       {xm, ym, zm} = {0.7946544722917661, 0.30353099910334297,
                       0.5257311121191336};
       φ𝕞 = ArcTan[ym/xm]; 𝕪𝕫 = ym/zm;
       ℙ1 = {xm, ym, 0.}; ℙ2 = {xm, ym, zm}; 𝕕 = #/Sqrt[#.#]&[ℙ2 - ℙ1];
(* map to symmetry unit *)
𝕗[s_, y_] := If[Chop[s] == 0., Pi/2., ArcTan[y/Sin[s]]];
toXYZ[{r_, φ_, s_}] = r {Cos[φ] Sin[#], Sin[φ] Sin[#],
                         Cos[#]}&[𝕗[s φ, ym/zm]]];

(* potential in the 1/120th part  *)
potentialφ[r_?NumberQ, φ_?NumberQ, s_?NumberQ] :=
Module[{rn = N[r], φn = N[φ], sn = N[s], ϑn}, ϑn = 𝕗[sn φn, 𝕪𝕫];
       icoφC[rn Cos[φn] Sin[ϑn], rn Sin[φn] Sin[ϑn], rn Cos[ϑn]]]
```

The next inputs generate an array of $\varphi(\{x, y, z\})$ values that later will be used to construct the equipotential surface.

```
(* define functions ρ1, ρ2In, and ρ2Out *)
SetOptions[FindRoot, MaxIterations -> 50];
With[{ε = 10^-6, δ = Sqrt[(5 + Sqrt[5])/10.]},
      (* make even contour surface value spacing *)
      (#1[c_] := r /. FindRoot[potentialφ[r, #2, #3] == c,
      {r, #4, #5}, Method -> Automatic])& @@@
      {{ρ1, 0, 0, 1/2, 2},
       {ρ2In, φm, 1, 1/2, 1 - ε}, {ρ2Out, φm, 1, 1 + ε, 2},
       {ρ3In, φm, 0, 1/2, δ - ε}, {ρ3Out, φm, 0, δ + ε, 2}}];

(* radial bounds for the equipotential surface *)
φMax = potentialφ[0, 0, 0];
rBounds[c_] := If[c <= φMax, {ρ1[c], ρ2Out[c]},
                 {Min[ρ2In[c], ρ3In[c]], ρ2Out[c]}]

(* 3×3×3 array of potential values *)
makeData[c_, pps_:{16, 16, 36}] :=
Module[{rMin, rMax}, {rMin, rMax} = rBounds[c];
       Table[potentialφ[r, φ, s], {s, 0, 1, 1/pps[[1]]},
             {φ, 0, φm, φm/pps[[2]]},
             {r, rMin, rMax, (rMax - rMin)/
                          Ceiling[pps[[3]](rMax - rMin)/1.3]}]]
```

The calculation of the surface parts from the potential data and the coloring of the surface is carried out next.

```
Needs["Graphics`ContourPlot3D`"]

(* various rotation matrices *)
(* inside a face of the icosahedron *)
Do[r[j] = {{1, 0, 0}, {0, Cos[j 2Pi/3], Sin[j 2Pi/3]},
           {0, -Sin[j 2Pi/3], Cos[j 2Pi/3]}} // N, {j, 0, 2}];

(* rotate into the position of other faces *)
With[{r = Table[C[k, l][i], {k, 3}, {l, 3}]},
Do[ℝ[i] = (r /. Solve[Table[r.ico[[3, 1, j]] == ico[[i, 1, j]], {j, 3}],
              Flatten[r]])[[1]], {i, 1, 20}]]

make120Parts[p_] := Table[Map[ℝ[j].#&, #, {-2}], {j, 20}]&[
                         Table[Map[r[j].#&, #, {-2}], {j, 0, 2}]&[
                                {p, Map[{1, 1, -1} #&, p, {-2}]}]]

(* color according to smallest distance from the wireframe *)
distanceColor[Polygon[l_]] :=
Module[{mp = Mean[l], h}, SurfaceColor[#, #, 2.6]& @
       Hue[2ArcTan[2.5 Sqrt[#.#]&[(# - #.𝕕 𝕕)&[mp - ℙ1]]]/Pi]]

(* make equipotential surface graphics for parameter c *)
equiφGraphics[c_, opts___] :=
Module[{part120, ℛ = rBounds[c], pr = 1.3 {{-1, 1}, {-1, 1}, {-1, 1}}},
(* φ == c in transformed coordinates *)
part120 = ListContourPlot3D[makeData[c],
              MeshRange -> {ℛ, {0, φm}, {0, 1}},
              Contours -> {c}, DisplayFunction -> Identity][[1]];
(* φ == c in Cartesian coordinates; make all 120 parts *)
Graphics3D[{EdgeForm[], {distanceColor[#],
            Map[ℛInv.#&, make120Parts[#], {-2}]}& /@
                Map[toXYZ, part120, {-2}]},
           opts, SphericalRegion -> True, PlotRange -> pr]]
```

Here are some of the resulting equipotential surfaces. For small values of *c*, the equipotential surface is basically spherical. Increasing *c* leads to dips in the faces of the icosahedron until φ^* is reached. A further increase leads to a closed surface with holes. Finally, for high values of *c*, the equipotential surfaces are smooth connections of tubes around the charged wire pieces along the edges of the icosahedron. The four values of *c* used in the following graphics are 24.1, 30.1, 33.338, and 34.31.

```
Show[GraphicsArray[equiφGraphics /@ #]]& /@
                                 {{24.1, 30.1}, {33.338, 34.31}}
```

For an animation, we do not use equidistant *c*-values, but calculate a set of 60 *c*-values such that the animation is as smooth as possible.

```
(* analyze potential values and partition in 60 intervals *)
contourφs = Module[{frames = 60, data},
data = Module[{pps = 30, ppφ = 20, ppr = 20, R = 1.3, ϕ = potentialφ, λ},
Table[ϕ[r, φ, s], {s, 0, 1, 1/pps}, {φ, 0, φm, φm/ppφ},
                    {r, 0, R, R/ppr}]];
      λ = Select[Sort[Flatten[data]], 24 < # < 35&];
     (* equal spacing of φ-values *)  #[[Round[Length[λ]/(2 frames)]]]& /@
                                   Partition[λ, Round[Length[λ]/frames]]];

(* generate frames for the animation *)
Do[Show[equiφGraphics[contourφs[[k]]], (* rotate viewpoint *)
        ViewPoint -> 1.8 {{ Cos[k/60 2Pi/5], Sin[k/60 2Pi/5], 0},
                          {-Sin[k/60 2Pi/5], Cos[k/60 2Pi/5], 0},
                          {0, 0, 1}}.{0.5, -0.36, 0.79}, Boxed -> False],
   {k, Length[contourφs]}];
```

It is also possible to implement larger programs inside a notebook instead of in a package. The next code implements a 3D Hilbert curve as an L-system. See [1335✱] for details. This time for the implementation, we use the typesetting capabilities of *Mathematica.* `"F"` moves forward, `"B"` moves backward and the other strings `"Ω"`, `"𝕋"`, `"𝒫"`, `"𝒬"`, `"£"`, and `"△"` implement various forms of turns.

```
HilbertCurve3D[n_Integer?Positive] :=
 Module[{axiom = "X",
   recursion = "X" → {"𝕋", "£", "X", "F", "𝕋", "£", "X", "F", "X",
      "𝒬", "F", "𝕋", "△", "△", "X", "F", "X", "Ω", "F",
      "𝒫", "△", "△", "X", "F", "X", "𝒬", "F", "△", "X", "𝒬", "△"},
   r = {0, 0, 0}, d = IdentityMatrix[3]},
  Prepend[DeleteCases[Which[(*the movements*)
       # == "F", r = r + (First /@ d),
       # == "B", r = r - (First /@ d);,
       # == "Ω", d = d.{{0, 0, 1}, {0, 1, 0}, {-1, 0, 0}};,
       # == "𝕋", d = d.{{0, 0, -1}, {0, 1, 0}, {1, 0, 0}};,
       # == "𝒫", d = d.{{0, -1, 0}, {1, 0, 0}, {0, 0, 1}};,
       # == "𝒬", d = d.{{0, 1, 0}, {-1, 0, 0}, {0, 0, 1}};,
       # == "£", d = d.{{1, 0, 0}, {0, 0, 1}, {0, -1, 0}};,
       # == "△", d = d.{{1, 0, 0}, {0, 0, -1}, {0, 1, 0}};,
       True, Null] & /@ Flatten[Nest[# /. recursion &,
                          Characters[axiom], n]], Null], {0, 0, 0}]]
```

Here are the points of a Hilbert curve of order 2.

```
hilbert = HilbertCurve3D[2]
```

Every point inside the cube with integer coordinates is touched exactly once by the Hilbert curve. Here is a quick check for this statement.

```
Sort[Flatten[Table[{i, j, k}, {i, 0, 3}, {j, 0, 3}, {k, 0, 3}], 2]] ==
 Sort[hilbert]
```

A graphic shows that the Hilbert curve winds through a cube.

```
hilbertLine = Line[HilbertCurve3D[2]];

Show[Graphics3D[{Hue[0], hilbertLine}], PlotRange → All, Axes → True]
```

Using a tube instead of a line shows more clearly, what the Hilbert curve looks like. The following code implements some functions generating a tube along a given line. The auxiliary routine `orthogonalDirections` constructs two orthogonal directions lying in the middle plane of the line segments `p1-p2-p3`. The auxiliary routine `prolongate` prolongates the point *p* of the tube along the direction *d*. Finally, the routine `tubify` generates a tube along the given line with a specified cross section.

```
orthogonalDirections[{p1_, p2_, p3_}] :=
  With[{n = #/Sqrt[#.#] &}, Module[{d}, If[Abs[#1.#2] == 1,
       If[Abs[#[[3]]] < 1, d = {-#[[2]], #[[1]], 0}, d = {0, #[[3]], -#[[2]]}],
 d = (#1 + #2) / 2]; n /@ {d, #1 × d}] &[ n[p3 - p2], n[p1 - p2]]];

prolongate[p_, q_, d_, {x_, y_}] :=
  Module[{s, u, v}, First[p + s d /. Solve[Thread[p + s d == q + u x + v y],
                   {s, u, v}]]];

tubify[Line[points_], startCrossSection_] :=
    MapThread[Polygon[Join[#1, Reverse[#2]]] &, #1] & /@
   Map[Partition[#, 2, 1] &, Partition[Rest[FoldList[Function[{p, t},
                                       (* propagate orthogonal system along the curve *)
        Module[{o = orthogonalDirections[t]},
         prolongate[#, t[[2]], (t[[2]]) - t[[1]], o] & /@ p]],
       startCrossSection, Partition[points, 3, 1]]], 2, 1], {2}];

startCrossSection[Line[l_], r_, n_] :=
 With[{p = (Position[l[[2]] - l[[1]], _?(# =!= 0 &), {1}, Heads → False][[1, 1]])},
  Table[l[[1]] + r Insert[{Cos[p], Sin[p]}, 0, p], {p, π/4, 9 π/4, 2 π/n}]]

addEnds[Line[l_]] := Line[Append[Prepend[l, 2 l[[1]] - l[[2]]], 2 l[[-1]] - l[[-2]]]]

hilbertLine = With[{m = Max[Transpose[hilbertLine[[1]]][[1]]]},
                    Map[#1 - {m, m, m} / 2 &, hilbertLine, {2}]];
```

Here is the above line “tubified”.

```
hilbertTube = tubify[N[addEnds[hilbertLine]],
   startCrossSection[hilbertLine, 0.25, 4]];

Show[Graphics3D[hilbertTube], PlotRange → All, Axes → False, Boxed → False]
```

Successively coloring the tube segments gives an even better idea of the Hilbert curve.

```
With[{l = Length[hilbertTube]},
    Show[Graphics3D[{EdgeForm[Thickness[0.001]], MapIndexed[
     {SurfaceColor[Hue[0.78 #2[[1]] / l], Hue[0.78 #2[[1]] / l], 2.1], #1} &,
     hilbertTube]}], PlotRange → All, Axes → False, Boxed → False]]
```

We make holes in the polygons to see through the long, dense tube spaghetti.

```
makeHole[Polygon[l_]] :=
    Function[m, MapThread[Polygon[Join[#1, Reverse[#2]]] &,
      {Partition[(Append[#, First[#]] &)[l], 2, 1],
       Partition[(Append[#, First[#]] &)[(m + 0.75 (# - m) &) /@ l], 2, 1]}]][
   Plus @@ l / Length[l]];

Show[%% /. p_Polygon :> makeHole[p]]
```

All of the above steps can be also made with a Hilbert curve of order 3.

```
hilbertLine = Line[HilbertCurve3D[3]];
```

Here are the values of the *x*-, *y*- and *z*-coordinates along the curve.

```
Show[Graphics[
         MapIndexed[
   {Hue[First[#2] / 5], Line[MapIndexed[{First[#2], #1} &, #1]]} &,
             Transpose[hilbertLine[[1]]]]], PlotRange → All, Frame → True]
```

Here is the Hilbert curve of order 3 in space.

```
Show[Graphics3D[{Hue[0], hilbertLine}], PlotRange → All, Axes → True]

hilbertLine = With[{m = Max[Transpose[hilbertLine[[1]]][[1]]]},
   Map[#1 - {m, m, m} / 2 &, hilbertLine, {2}]];
```

Here is a more circular tube along the Hilbert curve of order 3.

```
hilbertTube = tubify[N[addEnds[hilbertLine]],
   startCrossSection[hilbertLine, 0.25, 8]];

Show[Graphics3D[hilbertTube], PlotRange → All, Axes → False, Boxed → False]
```

Here is the colored Hilbert curve of order 3.

```
With[{l = Length[hilbertTube]},
 Show[Graphics3D[{EdgeForm[Thickness[0.001]], MapIndexed[
     {SurfaceColor[Hue[0.78 #2[[1]] / l], Hue[0.78 #2[[1]] / l], 2.1], #1} &,
     hilbertTube]}], PlotRange → All, Axes → False, Boxed → False]]
```

The cube containing the Hilbert curve is deformed into a sphere in the picture below.

```
toSphere[p_] :=
 p / √3 Function[q, Max[{q.#} & /@ {{1, 0, 0}, {-1, 0, 0}, {0, 1, 0},
        {0, -1, 0}, {0, 0, 1}, {0, 0, -1}}]][p / √p.p]

Show[Graphics3D[{EdgeForm[{Thickness[0.001], Hue[0.71]}],
        SurfaceColor[Hue[0.04], Hue[0.28], 2.12],
   Map[toSphere, N[hilbertTube], {-2}]}],
 PlotRange → All, Axes → False, Boxed → False]
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

1.3 What Computer Algebra and Mathematica 5.1

Can and Cannot Do

What *Mathematica* 5.1 Does Well

(~ by way of comparison with other programs):

■ Pattern matching

■ Symbolic integration

■ Numerical evaluation of the special functions of mathematical physics

■ Simplifying and calculating generalized hypergeometric functions

■ Solution of symbolic differential equations

■ Calculations with algebraic numbers

■ Numerical solution of differential equations

■ Symbolic summation

■ Implementing λ-calculus

■ Solving diophantine equations

■ Inequality solving

■ Quantifier elimination

■ Dealing with piecewise-defined functions

■ Graph Plotting

■ Allowing the development of large programs

■ Many other things

What *Mathematica* 5.1 Does Medium Well

Of course, *Mathematica* is not a perfect system. Here are things to improve (~ by way of comparison with a [skilled] human):

■ Integration of orthogonal polynomials

■ Calculations with Delta, Heaviside, and principal value distributions

■ Multidimensional symbolic integration

■ Solving transcendental equations

■ Series expansions at logarithmic and exponential singularities

…

Some more things to improve (~ by way of comparison with specialized programs):

■ Numerical solution of high-order univariate polynomials

■ Very large sparse eigenvalue problems

■ Global optimization problems

■ Noncommutative algebra

■ Calculating Schwarz-Christoffel mappings

■ Calculations within Clifford algebras

■ Calculating multiresultants

…

But check if a newer version of *Mathematica* can do some of the mentioned issues better.

What *Mathematica* 5.1 Cannot Do

Some pieces of (constructive) mathematics are not covered at all in the Version 5.1 of *Mathematica*, such as:

■ Analytically or numerically solving higher-order partial differential equations (especially elliptic ones)

■ Doing perturbation expansion of integrals, solutions of difference and differential equations

■ Solving eigenvalue problems for ordinary differential equations and systems

■ Numerical solution of stochastic differential equations

■ Solving functional equations

■ Fractional integration and differentiation

■ Solving integral equations

■ Solving Pfaff forms

■ Recognizing Painlevé transcendentals as solutions of differential equations and calculating them numerically

■ Calculating arbitrary transcendental functions of matrices

■ Displaying three-dimensional text in 3D graphics

■ Ray tracing and shadows in 3D graphics

■ Interpolation of surfaces with given boundary data

■ Calculating zeros of the special functions and their linear combinations (provided, to some limited extent, in the package `NumericalMath`BesselZeros``)

■ Calculating hypergeometric functions of several variables (to a limited extent with `AppellF1`)

■ *q*-versions of the hypergeometric, generalized hypergeometric, and confluent hypergeometric functions [55★] (however, see the package by C. Krattenthaler, *MathSource* 0206-705)

…

(The mentioned functionalities are not built-in, but can be, of course, implemented in *Mathematica*.

Again, check if a newer version of *Mathematica* is more capable here.

Many useful features can be added by packages, available in the standard package directory or from *MathSource,* http://www.mathsource.com. Maybe the reader will want to add something to these packages after becoming familiar with *Mathematica* as a language. Examples for notebooks still to be written include anholonomic constraints in classical mechanics [255★], [993★], [1159★], [1114★], [1260★], [590★], [904★], [310★], [452★], [1105★], [838★], visualization of the Bloch-Floquet theorem [423★], [52★], [536★], [617★], [778★], [1347★], [313★], [550★], [1370★], [707★], [1054★], [1420★], [1415★], [1416★], [436★], [902★], solving the Kohn-Sham equations [1051★], [403★], [1271★], [663✱], [578✱], [1344✱], [694✱], [752✱], [689✱], [693✱], [991✱], [1179✱], [48✱], [1099✱], [57✱], [1298✱], [893✱], [167✱], computation of Bell-like inequalities of higher order [59✱], [257✱], [107✱], [661✱], [1078✱], [7✱], [1365✱], [710✱], computation of Stokes constants for the asymptotics of linear differential equations [947✱], [337✱], [189✱], [938✱], [1424✱], [1048✱], [132✱], [658✱], [187✱], [336✱], [206✱], the recent renormalization group-based approach to asymptotic solution of differential equations [270✱], [1035✱], [1011✱], [802✱], [553✱], [430✱], [433✱], [1400✱], [1033✱], [1046✱], [803✱], [552✱], [434✱], [806✱], [804✱], computation of all special points and lines in a triangle [748✱], curvature induced bound states in tubes [410✱], [450✱] etc.

Some functionality that users might wish to have, but is not available in the Version 5.1 of *Mathematica*:

- Programmatic access to possible option values
- Full-fledged debugger
- Inactivating built-in evaluation and transformation rules
- Automatic code formatting

What *Mathematica* Is and What *Mathematica* Is Not

Without further comment, we include the following quotes concerning whether *Mathematica* (or more generally, computer algebra) is a useful tool for solving concrete problems. It is obvious, however, that for many application areas of mathematics, "experimental" mathematics [181✱], [180✱], [176✱], [71✱], [177✱], [1066✱], [70✱], [72✱], [1414✱], the applied sciences, engineering, finance, and other fields, computer algebra is a very useful tool.

> Two goals for PSEs [problem-solving environments, including computer algebra systems] are first, that they enable more people to solve more problems more rapidly, and second, that they enable many people to do things that they could not otherwise do.   from [513✱]

> The impact on mathematics of computer algebra and other forms of symbolic computing will be even larger than the impact of numeric computing has been.   from [295✱]

> The driving force in the 'eye of the hurricane' of technological and economic progress is and will be the finer and finer understanding of nature's structure and the more and more efficient use of the scientific technology of thinking, whose essence is mathematics and, today, self-automated mathematics.   from [209✱], [354✱]

But on the other hand, *Mathematica* (or a computer algebra system in general) does not solve all problems.

> Computer algebra is no substitute for mathematical creativity and mathematical knowledge; consequently, it is surely no universal mathematical problem solver. However, it makes the use of mathematical knowledge easier.   from [1231✱]

> … no computer algebra system can ever replace, in any significant way, mathematical thinking.   from [574✱]

> Of course, just as with paper and pencil calculations, the course of the evaluation [with a computer algebra system] must be guided with ingenuity and cleverness by the human mind behind the calculation.   from [402✱]

And *Mathematica* needs some learning time to use it efficiently.

> A computer algebra system is a tool, and the skill of the user is measured by the ability to turn the impossible into the trivial. In reality, this skill is rather easy to obtain. from [811✱]

For the mathematically inclined reader, *Mathematica* gives a lot of new opportunities.

> At a time when mathematicians are returning to computation, computers and symbolic computation programs are giving mathematicians an exciting opportunity to expand their research capabilities. from [815✱]

> They [computer algebra packages] provide extraordinary opportunities for research that most mathematicians are only beginning to appreciate and to digest. They also allow access to sophisticated mathematics to a very broad cross section of scientists and engineers. from [178✱]

Nevertheless, of the time it takes to learn *Mathematica*, the following remark is relevant for the rest of this book.

> … let us enjoy the present exciting transition era, where we can both enjoy the rich human heritage of the past, and at the same time witness the first crude harbingers of the marvelous computer-mathematics revolution of the late 21stcentury. from [1409✱] (see also [1410✱], [1412✱], [1413✱], [1153✱], and [1411✱])

> We only now are beginning to experience and comprehend the potential impact of computer mathematics tools on mathematical research. In ten more years, a new generation of computer-literate mathematicians, armed with significantly improved software on powerful computer systems, are bound to make discoveries in mathematics that we can only dream of at the present time. from [71✱]

However, we will not enter into a discussion about the relationship between mathematics and computations made possible by a computer. See [1061✱], [250✱] and references cited therein for this subject. Rather we will enjoy that "our generous universe comes equipped with the ability to compute" [63✱].

Exercises

1.$^{L?}$ What You Always Wanted to Compute

Find a problem or lengthy calculation (or a few of them) that you have always wanted to solve or carry out. Make a list of such problems, and as you read this book, try to find ways to solve your problems with *Mathematica*.

2.^{L2} *Mathematica* or Axiom or Maple or MuPAD or REDUCE or Form?

Compare the mathematical capabilities, clarity and uniformity of the syntax, required computational times, and directness with which mathematical ideas can be converted to programs using *Mathematica* and other (general-purpose) computer algebra systems. Use your computer if you have the technical (and financial) means. If not, page through, look at, read, and carefully study the corresponding handbooks and documentation. Here is some information on some of the various systems (for a more detailed listing and additional references, see [564✱]).

axiom:http://axiom.axiom-developer.org/

References: [679✱], [813✱]

Maple: http://www.maplesoft.com

References: [266✱], [267✱], [1138✱], [1001✱], [268✱], [535✱], [618✱], [777✱], [443✱], [444✱], [639✱], [515✱], [1116✱], [824✱], [687✱], [375✱], [700✱], [306✱], [307✱], [152✱], [642✱], [170✱], [572✱], [858✱], [859✱], and [99✱], as well as the Maple newsletters Maple Tech published by Birkhäuser and the newsgroup `comp.soft-sys.math.maple`.

MuPAD: http://www.mupad.de

References: [502✱], [525✱], [910✱], [1229✱], [886✱]

Reduce: http://www.uni-koeln.de/REDUCE/

References: [1296✱], [1241✱], [1242✱], [620✱], [877✱], and [1115✱]

Form: http://www.nikhef.nl/~form/

References: [1320✱], [1321✱]

For an overview of all general and special-purpose computer algebra systems and how to obtain them, see Computeralgebra-Report from Germany [301✱], http://SymbolicNet.mcs.kent.edu/systems/Systems.html, http://www.can.nl/-Systems_and_Packages/Per_Purpose/General/index_table.html.

For a standardized format (OpenMath) concerning the mutual exchange of data between computer algebra systems, see [1232✱], http://www.openmath.org/.

3.^{L1} Improvements?

We refer occasionally to some inconsistencies, restrictions, or bugs in *Mathematica* Version 5.1. If the reader has a newer version, check if these inconsistencies, restrictions, or bugs are still there, or if they have been removed.

Solutions

1. What You Always Wanted to Compute

A generic solution cannot be given for this exercise. If nothing occurs to you, here are a few suggestions that, after studying the references, can be more or less easily programmed in *Mathematica*. Some might look complicated at first glance, but they are not so complicated after some thinking about the subject; but some are not easy either.

a) Do noninteger derivatives such as $d^{\sqrt{2}}\, x^2 \exp(-x) \Big/ dx^{\sqrt{2}}$ exist? How are they defined? Are they unique? (For details, see [875✱], [1028✱], [1236✱], [1163✱], [228✱], [1408✱], [634✱], [389✱], [695✱], and [9✱].) A similar question would be: Are there fractional iterations, like $f(f(\ldots f(x)))$ (n f's, $n \in \mathbb{R}$)? (See [942✱], [672✱], [810✱], [569✱], [576✱], [23✱], [1387✱], [799✱], [1129✱], [1119✱], [24✱], [22✱], and [131✱].) Another similar one would be: Are there fractional finite differences? (See [690✱], [571✱], [1257✱], and [981✱].) For fractional differentials, see [314✱] and [271✱]. For fractional summation, see [974✱].

b) Is there a multivalued analytic function $f(z)$, where the value of the function on another sheet is just the derivative of the function on the principal sheet? (See [980✱], [588✱], [1345✱], and [125✱].) Are there functions $f(z)$ such that $\sum_0^\infty f(n) = \int_0^\infty f(z)\, dz$? (See [162✱], [1084✱], and [1227✱].) Which differential equations have solutions that are successive derivatives of some function? (See [321✱] and [322✱])

c) Is it possible to visualize the Banach-Tarski paradox? Loosely speaking, it is the creation of two oranges by slicing one into nonmeasurable pieces. (For details on the Banach-Tarski paradox, see [1334★], [492★], [758★], [1336★], and [335★].)

d) How fast should one run in rain (if caught without an umbrella) to keep as dry as possible? (For a solution based on an idealized box-person in homogeneous rain, see [347★], [350★], and [1040★]; for the properties of real rain, see [863★], [1133★], [1134★], [318★], [1064★], [1065★], [381★], and [1110★]; and for single rain drops, see [603★].)

e) How does one calculate puns (plays on words)? (See [148★], [149★], [1250★].)

f) Why are falling layers of water on fountains and waterfalls often wavy in the vertical direction? (See [243★] and [244★].)

g) What is the position of a regular-shaped piece of wood or other symmetric object floating in water? (See [357★], [1120★], [1274★], [532★], [688★], [446★], [1353★], [1354★], [1145★], and [93★]; for moving floating objects see [683★] and [1233★].) For the not unrelated problem of hanging pictures, see [160★].

h) Can one approximate locally a parametrically given curve better than via direct Taylor expansion in polynomials? (See [1103★], [1104★], [1178★], and [353★].)

i) How can Newton’s equation of motion be used to describe the movement of a bicycle (for simplicity, without a rider)? The problem involves a mechanical system with anholonomic side conditions. (See [158★], [601★], [484★], [211★], [67★], [471★], [157★], [1045★], [414★], [744★], [903★], [476★], [783★], [1044★], [993★], [842★], [1295★], [789★], [1256★], [1012★], [311★], [156★], [1030★], [517★], and [841★] and the references cited therein.) How does one express the closed form solutions of the equations of motions for the simplest nonholonomic systems— a rolling disk? (See [333★], [785★], [172★], [1323★], [171★], [801★], [1049★], and [1324★].) How does one describe the motion of a human symplectically? (See [666★].)

j) Taking into account air resistance, does a ball thrown straight up return earlier or later than without taking into account air resistance? (See [828★], [349★], [854★], and [1094★]; for a rotating ball see [499★], [500★], and [501★].)

k) What is the shape of a mylar balloon made from two circular sheets? (See [953★], [1058★], and [952★]; for inflating rubber balloons, see [973★]; for larger balloons, see [64★].)

l) What model would be used for a falling cat that always lands on its legs? (For a model cat solution, see [701★], [961★], [497★], [853★], [1091★], [459★], and [962★].) At which position does a falling tower brake? (See [879★] and [1310★].)

m) Why does sand in shallow sea water have ridges? What determines the wavelength and height of these ridges? (For an appropriate model, see [1006★], [878★], and [338★]. Concerning the outside-water behavior of sand, see [932★].)

n) How does one derive the scaling relation between the mass and the metabolic rate of an animal or plant? (See [406★] and [1366★].) How many different animal species could exist? (See [1293★].)

o) Can one calculate closed-form expressions for the gravitational potential and the moment of inertia of the regular polyhedra? (See [986★], [1339★], [786★], [577★], [596★], [1340★], [130★], [847★], [1276★], [74★], [1364★], [1197★], [816★], [585★], and [253★]. For polarizabilities of Platonic solids, see [1211★].)

p) Can one calculate how a piece of paper tears? (See [1026★], [1161★].) (For crumpling of paper, see [382★], [1199★], [909★], [544★], [21★], [383★], [397★], [1385★], [1162★], [845★], and [1074★]; for wrinkling, see [256★].)

q) Given the path of the front wheels of a car, what is the path of the rear wheels? (See [490★], [1317★], [540★], [1275★], [220★], and [943★]. For four steerable wheels, see [1402★]. For bicycle tracks, see [469★]. For towing, see [1121★].)

r) Given a square with integer side lengths, is it possible to tile this square into triangles so that all triangle side lengths again are an integer? (See [594★].) And what is the largest square that can be inscribed in a unit cube? (See [325★].)

s) Can one model how a piece of paper (or a leaf) falls? (See [1063★], [1263★], [43★], [465★], [722★], [882★], [913★], and [475★].)

t) How high can a given kite at given wind and with given cord length fly? (See [1372★].)

u) What is the apparent form of a train moving with relativistic velocity? (For the appearance of fast moving simple geometric bodies, see [1361★], [10★], [161★], [633★], [604★], [646★], [1386★], [1270★], [198★], [1152★], [464★], [890★], [1113★], [791★], [463★], [60★], [995★], [1004★], [1360★], [141★], [1010★], and [685★].) And do very fast large cars ($v \approx 0.9\ldots9\ c$, c the velocity of light) fit in short garages because of length contraction? (See [518★], [1200★], and [324★].) For submarines, see [911★].

v) What is the probability that a thick coin will fall on its side if dropped randomly? (See [169★], and [719★].)

w) How does a tippe top work? (See [183★], [1014★], [659★], [1228★], [293★], [1082★], [839★], [425★], [1036★], [807★], [94★], and [956★].) For spinning cooked eggs, see [1167★].

x) What is the form of the closed plane curve of greatest possible area that can be moved around a right-angled corner in a hallway? (See [528★].)

y) Does a buttered slice of toast land with the buttered side down really more often? (See [915★], [62★], and [432★].)

z) How is the scale of a sundial determined? (See [1176★], [1160★], [650★], [1300★], [1175★], [1188★], [1351★], [1418★], [1147★], [504★], [154★], and http://www.mathsource.com/cgi-bin/MathSource/0209-001.)

a′) How can the growth of icicles be modeled? (See [1020★], [736★], [888★], and [1019★]. For similar patterns on water columns, see [1217★]. For modeling snowflakes, see [846★], [819★], [1319★], [676★], and [677★].)

b′) Mathematically, how does a queue of cars (on the freeway) form, and how long does one have to wait in line (as a function of the parameters traffic density, average speed, etc.)? (For mathematical models of traffic flow, see, e.g., [1369★], [1149★], [1172★], [623★], [883★], [283★], [503★], [323★], [404★], [405★], [1380★], [153★], [800★], [984★], [1170★], [985★], [493★], [1168★], [621★], [1169★], [482★], [526★], [1187★], [867★], [481★], [684★], and the references cited therein. For modeling the driver's experience, see [1117★]. For pedestrian traffic, see [218★], [649★], [1171★], [219★], [624★], [755★], [895★]. For the modeling of the corrugation of roads, see [182★].)

c′) How does one algorithmically measure k gallons given n jugs with given capacities? (See [168★].)

d′) Which point of a hypercube in n dimensions maximizes the product of the distances to its vertices? (See [1371★].)

e′) When leaves fall from the trees in the autumn, assume that all of the ground is covered by leaves. How many leaves does one in average see inside a certain area? (See [320★], [396★], [558★], and [790★] for circular leaves.) A related, but easier problem is: What is the average height children will pile rectangular blocks while building towers before they collapse? (See [671★].)

f′) How does one model a dripping tap? (See [1186★], [498★], [316★], [712★], [760★], [657★], [315★], [31★], [1127★], [1126★], [387★], and [208★], [761★].)

g′) How does one calculate the shape of a water drop on a smooth surface? (See [204★], [1015★], [1140★], [103★], [798★], [1★], [1267★], [817★], and [900★]; for moving drops, see [30★].)

h′) How does one describe the motion of a curling rock? (See [1201★], [680★], [365★], [1203★], [1204★], [1060★], [364★], [1202★], and [457★].) How does one model stones skimming over water? (See [165★]). How does one model the increasing frequency of the whirring sound of a coin rotating on a table? (See [954★], [955★], [730★], [1067★], [424★], [146★], and [1238★].)

i′) How does one calculate the optimal form of the teeth of gears? (See [855★], [1392★], [193★], [1346★], and [1086★].)

j′) How does one model a Levitron®? (See [135★], [1212★], [559★], [415★], [1213★], [523★], and [136★]; for nonlinear levitation, see [946★].) How can one model the woodpecker toy? See [1069★] and [826★].

k′) How does one model the shape of a human trail system on a meadow? (See [622✶].) (For modeling the flow going out of a large hall, see [1261✶], [194✶], [664✶]; for modeling standing, see [407✶]; for ski slopes, see [429✶].)

l′) Can two losing games yield a winning game? (See [606✶], [386✶], [47✶], [1265✶], [1278✶], [1316✶], [265✶], [948✶], [607✶], [936✶], [937✶], [472✶], [1112✶], [732✶], [717✶], [25✶], [945✶], and [1052✶].)

m′) How does a grooved cylinder roll down an inclined plane? (See [931✶].) How does one model the "Indian rope trick"? (See [4✶], [1038✶], [5✶], [652✶], [972✶], and [262✶].)

n′) How does one calculate the shape of the two pieces used to cover of baseball? (See [1273✶].)

o′) How does one model the learning of grammar? (See [1009✶].)

p′) How does one describe the path of a single air bubble rising in water? (See [1395✶], [971✶], [376✶], [1308✶], [641✶], [1089✶], and [848✶].)

q′) Which numbers can be expressed in a closed form? (See [281✶] and [98✶].) And what numbers are computable? (See [1359✶].)

r′) Can one use the logistic map to generate random numbers? (See [33✶], [545✶], [546✶], [1384✶], [547✶], [1299✶], and [453✶].)

s′) What are the side lengths of a rectangle with a given maximal area, such that the area/perimeter ratio is as large as possible? (See [905✶].)

t′) How does one model a continuous transition from Taylor series coefficients to Fourier series coefficients? (See [54✶], [1077✶], and [1076✶].)

u′) How does one model the waiting time for a web browser connection? (See [1374✶], [18✶], [19✶], [1262✶], [77✶], [918✶], [648✶] and [836✶], [743✶] for the cables.)

v′) Is there a multidimensional version of Simpson's rule? (See [644✶].)

w′) What is the (continuous) symmetry of the genetic code? (See [643✶], [92✶], [486✶], [91✶], [478✶], [487✶], [39✶], [40✶], [417✶], [1193✶], [488✶], [1194✶], [686✶], [992✶], [491✶], and [727✶].)

x′) Are there functions whose reciprocal is equal to their inverse? (See [272✶].) How to calculate the Fourier coefficients of the reciprocal function from the Fourier coefficients of a function? (See [413✶].)

y′) Is there a linkage that signs your name? (See [705✶], [749✶], [750✶], [751✶], [155✶], [531✶], [458✶], and [399✶].)

z′) How does one model bird and fish swarms? (See [334✶] and [957✶].)

a″) How does one model the various gaits of a horse? (See [298✶], [299✶], [1303✶], [1304✶], [213✶], [929✶], [529✶], and [1248✶]; for human gait modeling, see [1258✶], [419✶], [769✶], and [1318✶].) How to classify juggling patterns? (See [210✶], [431✶], [1085✶], and [1237✶].)

b″) How does one model the shape of a cracking whip? (See [551✶], [930✶], and [793✶].)

c″) What is the probability of going to jail in the Monopoly ® game? (See [1393✶].)

d″) How does one construct a computable bijection between the rational numbers and the integers (the classical diagonal method is not easy to compute for reduced fractions)? (See [1047✶] and [248✶].)

e″) How does one model collapsing bridges? (See [927✶].)

f″) How does one model the movement of a camphor scraping on water? (See [988✶], [989✶], [982✶], and [616✶].)

g″) Given a rectangle, how many congruent rectangles can you position around it such that each one touches the given rectangle, but does not intersect with any of the others? (See [723✶] and for polyhedra [1342✶].)

h″) What are the possible equilibrium shapes for closed elastic rods? (See [818★], [1331★], [914★], [670★], [998★], and [49★].)

i″) How does one model the movement (and potential self-knotting) of a moving hanging chain? See ([106★] and [233★]).

j″) How many fingers form an “optimal” hand? (See [950★], [756★], [901★], and [949★].)

k″) How many different ancestors do humans have on average in their genealogical tree? (See [368★], [369★], [370★], [359★], [1218★], [1024★], [1222★], [1266★], and [987★].) (And how does one model the shape of the phylogenetic tree? See [912★], [147★], and [1243★]. For the related problem: the distribution of family names, see [1407★], [891★], [304★], [1118★], and [662★].)

l″) Are the magnetic field lines around a current-carrying wire really closed? (See [1219★], [1279★], [377★], [1068★], [1158★], problem 18 of [1297★], and [1073★].)

m″) How does one calculate polynomials orthogonal over a regular polygon? (See [1422★].)

n″) On which day of the week should a teacher hold an exam to maximize the surprise when it happens? (See [282★] and [1221★].)

o″) How does one model river basins? (See [1135★], [390★], [391★], [224★], and [392★].)

p″) How does one model a ball rolling on a rough surface? (See [1311★].)

q″) What is the probability for a random walker in d dimensions to return to the origin? (See [108★] and [109★].)

r″) How does one model the expansion of a popcorn kernel? (See [640★] and [1102★].)

s″) What is the explicit form of the eigenfunctions of the curl operator? (See [1406★], [231★], [965★], [968★], and [1071★].) For the exponential of the curl operator, see [1088★]; for the discretized version, see [285★].

t″) How does one model the bubbling of wine bottle labels? (See [200★].)

u″) How does one analytically map a polygon with a hole to an annulus? (See [780★], [779★], and [358★].)

v″) How does one construct and model a gravity-powered toy that can walk but not stand? (See [296★] and [297★].)

w″) How does one represent a function of several variables as a superposition of functions of one variable? (See [496★], [15★], and [534★].)

x″) Why can dolphins swim so fast? (See [852★].)

y″) Can a band-limited function oscillate faster than its bandwidth? (See [11★], [725★], [726★], [225★], [1100★], [134★], [133★], [12★], [13★], and [724★].) For the definition of an instantaneous frequency, see [1031★].

z″) How does one straighten out a chain of connected rods in three and four dimensions? (See [145★] and [290★].)

a‴) How does one model the generation and sound of canary songs? (See [520★], [1277★], and [753★]; for the modeling of snoring, see [14★].)

b‴) How does one model the sand flow in a hourglass? (See [462★].)

c‴) How does one model the consequences of increasing information exchange on inter-personal interactions? (See [1419★].)

d‴) How does one cut out any planar straight-line figure from one sheet of paper with a single straight cut? (See [361★] and [1039★].)

e‴) If integration is the limit of a sum, what is the corresponding limit for a product? (See [395★], [533★], [706★], [32★], [602★] and [1181★] for matrices.)

f‴) How densely can Platonic solids be packed on a lattice? (See [140★].)

g‴) Given a polynomial with complex roots only, what is the "nearest" polynomial with a real root? (See [635★].)

h‴) Are there nonlinear differential equations whose solutions obey a superposition principle? (See [1376★], [1210★], [1291★], [214★], [215★], [549★], [978★], [237★], [129★], [735★], [236★], [1062★], and [235★].)

i‴) How "quadratic" are the natural numbers? (See [1146★].)

k‴) How does one mathematically discriminate between a novel and a poem? (See [44★], [302★], and [303★].)

l‴) How does one calculate the ideal steak cooking time and flipping times? (See [925★], [1154★], [1032★], and [81★].)

m‴) How does one effectively fight a hydra that regrows its heads? (See [754★], [609★], and [869★])

n‴) Can one eliminate all variables from a symbolic calculation? (See [1246★], [332★], and [1018★].)

o‴) How does one model the noise of helicopter blades? (See [456★], [455★], [241★], [605★], [864★], and [197★]; for the squeal of train wheels, see [619★]; and the sound of rubbing hands, see [1403★].)

p‴) How does one experimentally measure and mathematically model a Riemann surface? (See [1195★].)

q‴) Given the first terms of a Taylor series, how does one recover the original function? (See [384★] and [651★].)

r‴) What is the probability to encounter a matrix difficult to invert? (See [362★].)

s‴) How does one model folded proteins? (See [1377★], [1264★], [75★], [201★], and [27★].)

t‴) How frequently does a given word or phrase statistically appear as a subsequence in a text? (See [470★].)

u‴) How does one model the creation of aeolian sand ripples? (See [331★], [1405★], [1008★], [628★], [796★], [797★], [418★], [630★], [1189★], [1007★], [941★], [34★], [35★], [849★], [958★], and [629★].)

v‴) How does one calculate all possible tie knots? (See [468★].)

w‴) Can calculations exhibit phase transitions? (See [638★], [665★], [959★], [939★], [409★], [260★], [933★], [1398★], [1399★], [1358★], [757★], [873★], [934★], [87★], [195★], [291★], and [1357★].) (For phase transitions in the World Wide Web, see [144★]; for phase transitions in data compression, see [977★]; for phase transitions in parameter-dependent wave functions, see [667★].)

x‴) What is the expected average of the chord length of random lines intersecting a closed plane curve? (See [924★].)

y‴) How does one dissect a polygon into polygonal pieces that are connected by flexible hinges and allow to form the mirror image of the original polygon? (See [445★].)

z‴) How does a prismatic cylinder roll down an inclined plane? (See [1254★] and [2★].)

a$^{\text{iv}}$) How many zeros does a random trigonometric polynomial have in average? (See [454★].)

A host of other suggestions, both large and small, can be found almost daily in the newsgroup rec.puzzles http://dejanews.com, http://star.tau.ac.il/QUIZ, http://problems.math.umr.edu and related websites (http://dmoz.org/-Science/Math/Mathematical_Recreations contains a listing of such websites). We also mention the *American Journal of Physics,* http://www.amherst.edu/~ajp, and *European Journal of Physics*, and Eric Weisstein's MathWorld http://mathworld.wolfram.com (Concise Encyclopedia of Mathematics [1363★]), the *Journal of Recreational Mathemat ics* as well as http://www.seanet.com/~ksbrown. (See also [1247★].)

For the more theoretical physics-interested reader, we mention a few more technical possibilities.

α) How does one construct (pseudodifferential) cube roots from a differential operator (similar to $\gamma^\mu\,\partial_\mu + m$ is a square root of $\partial_\mu \partial^\mu + m^2$)? (See [729★], [728★], and [1083★].) For square roots of the heat equation, see [1314★].

β) How does one construct $p+1$ orthonormal bases in a p-dimensional vector space over ℂ, such that all possible scalar products between vectors from different bases have the same magnitude? (See [46★], [1388★], [1389★], [764★], [121★], [1151★], [1391★], [763★], [1125★], [1079★], [669★], [441★], [269★], [45★], [1330★], [355★], [76★], [1379★], [122★], [53★], and [1390★].)

γ) In how many different orthogonal coordinate systems is the wave equation separable? (See [696★], [697★], [124★], and [1337★].)

δ) Is there a potential $V(x)$, such that the eigenvalues of the corresponding one-dimensional Schrödinger equation are the prime numbers? (See [979★] and [1381★].) (For the related problem of a potential whose eigenvalues are the imaginary parts of the nontrivial zeros of the Riemann Zeta function, see [246★], [822★], [1394★], [1309★], [1150★]; for the Jost function having the zeros of the Riemann Zeta function, see [737★] and [738★] and for potentials that represent the prime numbers, see [366★].)

ϵ) Given two hermitean matrixes $\boldsymbol{K}$ and $\boldsymbol{L}$, what can be said about the spectrum of $\boldsymbol{K}+\boldsymbol{L}$? (See [771★], [772★], [773★], [774★], [339★], and [506★].) What about the spectrum of $\boldsymbol{K}.\boldsymbol{L}$? (See [1350★].) Given two polynomials $p(x)$ and $q(x)$, what can be said about the factorization of $p(x)+q(x)$? (See [746★].)

ε) How fast is the “ultimate laptop”? (See [860★], [999★], [1000★], [861★], [483★], [1056★], [227★], [1234★], [792★], and [1041★].) (For space-time possibilities to speed up computations, see [422★], [447★], [1037★], and [205★]; for superluminal methods, see [1225★]; for limits on the hard drive capacities, see [104★], [292★].)

ζ) How does one generate Greechie diagrams efficiently? (See [926★].)

η) Can one model a DLA cluster deterministically? (See [610★], [840★], [344★], [345★], [83★], [84★], [625★], [85★], and [82★].)

θ) Are there (sensible) nonhermitian Hamiltonians with real spectra? (See [119★], [1425★], [970★], [767★], [969★], [1356★], [66★], [401★], [110★], [229★], [114★], [115★], [38★], [112★], [230★], [113★], [112★], [461★], [116★], [117★], [940★], [118★], [111★], and [120★].)

ϑ) How does one model the movement of an adiabatic movable piston between two gases in equilibrium? (See [278★], [583★], [581★], [731★], [935★], [823★], [892★], [222★], [327★], [582★], [276★], [328★], [1367★], [329★], [277★], [963★], [279★], [1070★], [330★], [199★], and [994★].)

ι) Can knots be stable solutions of classical field theories? (See [96★], [1002★], [1023★] and [1136★].) And can knots be formed by the zero lines of hydrogen wave functions? (See [137★].)

κ) How does one (numerically) calculate the length and the dimension of the path of a quantum particle? (See [795★], [543★], and [794★].)

$\varkappa$) How does a rope or chain slide off the edge of a table? A little contemplation of the conservation of momentum law shows immediately that the standard solution from experimental physics books is wrong. (For details, see [1165★], [363★], [1092★], [102★], [186★], and [1333★]; for folded chains, see [1245★].)

λ) How does one “properly” discretize Maxwell’s equations? (See [1268★], [916★], [708★], [579★], [917★], [1307★], and [745★].) For superconsistent discretizations in general, see [507★]. Are there oscillating charge distributions that do not radiate? (See [514★], [494★], [539★], [899★], [637★], [745★], [374★], [1005★], and [245★].)

μ) Can bend cylinders support bound states (in a quantum-mechanical sense)? (See [653★], [656★], [1029★], [862★], [410★], [37★], [542★], [411★], [570★], [238★], [239★], [240★], [1226★], [1087★], [881★], [951★], and [508★].) For additional linking, see [747★]. Can neutral multipole arrangements of charges support bounds states? See [1097★] for the general case and [1292★], [668★], [1072★], [1057★], [1130★] for dipols.

ν) Can one model any one-dimensional contact interaction with delta function potentials (in a quantum-mechanical sense)? (See [17★], [477★], [1207★], [20★], [317★], [275★], [682★], [1294★], [1050★], [1184★], [451★], [1396★], [788★], [1208★], [1209★], [808★], [505★], [983★], [1287★], [1343★], [1288★], [273★], [26★], [1289★], and [274★].)

ξ) What is the efficiency of a Carnot machine that uses an ideal Bose gas or Fermi gas? (See [1214★], [1215★], and [1216★].)

o) How does one construct the quantum mechanical hydrogen wave functions from classical orbits? (See [716★], [714★], and [715★].)

ϖ) How does one calculate terms of the Rayleigh–Schrödinger perturbation theory when the integrals in $\langle i \mid V \mid j\rangle$ diverge? (For instance $V(x) \sim \exp(x^4)$ in the harmonic oscillator basis?) (See [765★], [373★], [573★], [352★], [766★], [608★], [885★], and [287★].)

π) Can one observe the spin of a free electron in a Stern-Gerlach-type experiment? (See [95★] and [521★].)

ρ) How does one stabilize classical mechanics? (See [1325★], [627★], [1326★], [97★], and [420★].) (For the dequantization of quantum mechanics, see [660★]. For the fundamental constants of classical mechanics, see [928★].)

ϱ) What is the connection between Huygens' principle with the wave equation? (Huygens' principle states that from every point on a wave, a spherical basic wave emerges there.) Is it possible to model the spreading out of a wave directly from Huygens' principle numerically? (See [69★], [342★], [105★], [341★], [289★], [440★], [589★], [247★], [1021★], [159★], [655★], [827★], [1423★], and [1375★].)

σ) Do one-dimensional lattices show Fourier's law in heat conduction? (See [837★], [1095★], [647★], [41★], [42★], [428★], [843★], [378★], [379★], [522★], and [530★].)

ς) How does one formulate classical mechanics using a Hilbert space? (See [921★], [784★], [560★], [561★], [562★], [919★], [8★], [284★], [192★], and [191★]. For a path integral formulation, see [920★], [3★], and [563★]; for a Wigner distribution, see [512★]; and for a unifying approach, see [759★].)

τ) What is the relation between a d-dimensional Kepler problem with a $2d-2$ dimensional harmonic oscillator problem? (See [1081★], [1269★], [1426★], [814★], [698★], [343★], [739★], [309★], [996★], [305★], [897★], [254★], [86★], [699★], [1417★], [88★], [234★], [896★], and [740★].)

υ) Are there stable atoms in d dimensions and how does the corresponding periodic table look like? (See [216★], [894★], [856★], [1109★], [631★], [591★], [1198★], [58★], [741★], [742★], [787★], [809★], and [692★].)

ϕ) How does one calculate the numerical value of the Boltzmann constant k_B? (See [782★], [437★], and [825★]. For the experimental determination, see [474★]; for the status as a constant, see [412★].)

φ) How does one model the wave function of a photon emitted from an excited atom? (See [966★], [967★], [721★], [221★], [264★], [527★], [251★], [356★], [832★], [143★], [6★], [473★], and [898★]. For absorbing a photon, see [61★]. For localizing photons, see [1156★].)

χ) How does one calculate higher-order Foldy–Wouthuysen transformations? (See [964★], [1306★], [1122★], [1124★], [805★], [421★], [466★], [89★], [90★], [734★], [1123★], and [1003★].)

ψ) Is the charge distribution of a finite one-dimensional wire uniform? (See [674★], [575★], [673★], [348★], [1206★], [16★], [720★], [1301★], and [718★].)

ζ) What are the crystal classes in 4D? (See [216★], [1034★], [1368★], [202★], [212★], [997★], [1383★]; [1322★], [1230★], [1253★] for 5D; and [1190★], [1191★], and [626★], [1107★] for nD.)

ω) How does a light beam behave in a water vertex? (See [833★], [1329★], [835★], [73★], [1332★], [944★], [851★], [80★], and [834★].)

$\digamma$) How does one calculate the electromagnetic field of a charge moving above a conducting surface? (See [188★], [1174★], [1043★], and [1173★]; for a corrugated surface, see [1302★] and [597★]; for an array of half-planes, see

[812★]; for moving current loops, see [1042★].) What is the magnetic field around a magnet moving through a metallic tube? See [1053★].

ϙ) How does one model physical systems with negative specific heat? (See [876★].)

ϛ) Are there 2D potentials with two families of orthogonal trajectories? (See [1098★].)

ϡ) What are the eigenvalues of a (grand) canonical density matrix? (See [263★], [1027★], [733★], [495★], [286★], and [1290★].)

α') What is the relativistic generalization of the Ampere–Maxwell law in integral form? (See [960★], [524★].)

β') What is the relativistic generalization of the Fokker–Planck equation for Brownian motion? (See [416★].)

γ') Is there a fluctuation theorem for a single anharmonic oscillator? (See [1183★].)

δ') In how many different orders can k spacelike separated events in Minkowski space be observed? (See [1239★].)

ϵ') What is the average shape of a random walk in many dimensions? (See [1155★].)

ε') What is the nature of entanglement in a free electron gas? (See [1022★], [871★], [872★], and [1315★].)

ζ') How to quantify statistical properties of thermodynamic fluctuations? (See [880★], [326★], [1132★], [678★], [923★], [448★], [449★], and [866★].)

(For a set of more advanced problems, see [595★], [1373★], [675★], [29★], [325★], [203★], [101★], and http://www.math.princeton.edu/~aizenman/OpenProblems.iamp. For more advanced computational geometry problems, see http://www.cs.smith.edu/~orourke/TOPP/). For the “big” problems, see [1101★] and http://boudin.fnal.gov/NNP/-B1798866615/ .

2. *Mathematica* or axiom or Maple or MuPAD or REDUCE or Form ?

This cannot be answered here. It depends largely on what you require from a computer algebra system. For the opinions of several reviewers, see the references listed in the Appendix. You should make an informed decision yourself whether *Mathematica* is the correct system for your special applications. Some of the things that can be done with *Mathematica* will be shown in the following chapters of this book.

At the time this book was written, a good (objective) indication existed that *Mathematica* is the right choice for the reader. It was the only system that was able to solve all of the ten (easy to state, but not so easy to solve) problems from the 1997 ISSAC [International Symposium on Symbolic and Algebraic Computation] system challenge [1282★], http://www.wolfram.com/news/archive/issac. The summary of the challenge session states: “… there can really be only one choice. Only one team correctly solved all problems. Only one team solved every problem in more than one way as a check for their solution. … The team was *Mathematica*’s team.” [308★].

A more recent, and more hard-core numerical oriented, problem set was Nick Trefethen’s 100$–100-digit challenge [1280★], [173★]. Comparing the solutions and the solution techniques employed by users of a variety of programs [1281★] shows that frequently *Mathematica* allowed for the most straightforward, shortest, most elegant solutions, frequently even in a symbolic form using the special functions of mathematical physics (which are discussed in the Symbolics volume [1285★] of the *GuideBooks*). (And again, the *Mathematica* team, among others, was be able to solve all problems correctly.
(See http://web.comlab.ox.ac.uk/oucl/work/nick.trefethen/hundred.html for details.)

And although we cannot ask David Hilbert directly anymore, G. J. Chaitin says “I think that Hilbert would have loved *Mathematica* … because in a funny way it carries out Hilbert’s dream, as much as it was possible.” [259★].

3. Improvements?

Just try it!

References

✱1 S. Abe, J. T. Sheridan. *Phys. Lett.* A 253, 317 (1999). *DOI-Link*

✱2 R. Abeyaratne. *Int. J. Mech. Eng. Educ.* 17, 53 (1989).

✱3 A. A. Abrikosov, Jr., E. Gozzi, M. Mauro. *Mod. Phys. Lett.* A 18, 2347 (2004). *DOI-Link*

✱4 D. Acheson. *Proc. R. Soc. Lond.* A 443, 239 (1993).

✱5 D. Acheson. *From Calculus to Chaos*, Oxford University Press, Oxford, 1997. *BookLink (2)*

✱6 C. Adlard, E. R. Pike, S. Sarkar. *arXiv:quant-ph*/9707027 (1997). *Get Preprint*

✱7 D. Aerts, S. Aerts, J. Broekaert, L. Gabora. *arXiv:quant-ph*/0007044 (2000). *Get Preprint*

✱8 D. Aerts, B. Coecke, B. D'Hooghe, F. Valckenborgh. *arXiv:quant-ph*/0111074 (2001). *Get Preprint*

✱9 R. Agarwal, M. Bohner. *Result. Math.* 35, 3 (1999).

✱10 J. M. Aguirregabiria, A. Hernandez, M. Rivas. *Am. J. Phys.* 60, 597 (1992). *DOI-Link*

✱11 Y. Aharonov, J. Anandan, S. Popescu, L. Vaidman. *Phys. Rev. Lett.* 64, 2965 (1990). *DOI-Link*

✱12 Y. Aharonov, N. Erez, B. Reznick. *arXiv:quant-ph*/0110104 (2001). *Get Preprint*

✱13 Y. Aharonov, N. Erez, B. Reznik. *Phys. Rev.* A 65, 052124 (2002). *DOI-Link*

✱14 T. Aittokallio, M. Gyllenberg, O. Polo. *Math. Biosci.* 170, 79 (2001).

✱15 S. Akashi. *Bull. Lond. Math. Soc.* 35, 8 (2003). *DOI-Link*

✱16 A. D. Alawneh, R. P. Kanwal. *SIAM Rev.* 19, 437 (1977).

✱17 S. Albeverio, L. Dabrowski, S.-M. Fei. *arXiv:quant-ph*/0001089 (2000). *Get Preprint*

✱18 R. Albert, H. Jeong, A.-L. Barabási. *arXiv:cond-mat*/9907038 (1999). *Get Preprint*

✱19 R. Albert, H. Jeong, A.-L. Barabási. *Nature* 401, 130 (1999). *DOI-Link*

✱20 S. Albeverio, S.-M. Fei, P. Kurasov. *arXiv:quant-ph*/0206112 (2002). *Get Preprint*

✱21 R. F. Albuquerque, M. A. F. Gomes. *Physica* A 310, 377 (2002). *DOI-Link*

✱22 P. Aldrovandi, L. P. Freitas. *arXiv:physics*/9712026 (1997). *Get Preprint*

✱23 R. Aldrovandi, L. P. Freitas. *J. Math. Phys.* 39, 5324 (1998). *DOI-Link*

✱24 R. Aldrovandi. *Special Matrices of Mathematical Physics*, World Scientific, Singapore, 2001. *BookLink*

✱25 A. Allison, D. Abbott. *arXiv:cond-mat*/0208470 (2002). *Get Preprint*

✱26 V. Alonso, S. De Vincenzo. *Int. J. Theor. Phys.* 39, 1483 (2000). *DOI-Link*

✱27 J. L. Alonso, G. A. Chass, I. G. Csizmadia, P. Echenique, A. Tarancón. *arXiv:q-bio.BM*/0407024 (2004). *Get Preprint*

✱28 N. Altshiller-Court. *Modern Pure Solid Geometry*, Macmillan, New York, 1935. *BookLink*

✱29 A. Amann, U. Müller–Herold in H. Atmanspacher, A. Amann, U. Müller–Herold (eds.). *On Quanta, Mind and Matter*, Kluwer, Dordrecht, 1999. *BookLink*

✱30 M. B. Amar, L. J. Cummings, Y. Pomeau. *Physics Fluids* 15, 2949 (2003). *DOI-Link*

✱31 B. Ambravaneswaran, S. D. Phillips, O. A. Basaran. *Phys. Rev. Lett.* 85, 5332 (2000). *DOI-Link*

✱32 P. K. Andersen, Ø. Borgan, R. D. Gill, N. Keiding. *Statistical Models Based on Counting Processes*, Springer-Verlag, Berlin, 1993. *BookLink (2)*

✱33 M. Andrecut. *Int. J. Mod. Phys.* B 12, 921 (1998). *DOI-Link*

✱34 B. Andreotti, P. Claudin, S. Douady. *arXiv:cond-mat*/0201103 (2002). *Get Preprint*

✱35 B. Andreotti, P. Claudin. *arXiv:cond-mat*/0201105 (2002). *Get Preprint*

✱36 G. E. Andrews. *The Theory of Partitions*, Cambridge University Press, Cambridge, 1998. *BookLink (3)*

✱37 M. Andrews, C. M. Savage. *Phys. Rev.* A 50, 4535 (1994). *DOI-Link*

✱38 A. A. Andrianov, F. Cannata, J.-P. Dedonder, M. V. Ioffe. *arXiv:quant-ph*/9806019 (1998). *Get Preprint*

✱39 F. Antoneli, Jr., L. Braggion, M. Forger, J. E. M. Hornos. *Int. J. Mod. Phys.* B 17, 3135 (2003). *DOI-Link*

✱40 F. Antoneli, Jr., M. Forger, J. E. M. Hornos. *Mod. Phys. Lett.* B 18, 971 (2004). *DOI-Link*

✱41 K. Aoki, D. Kusnezov. *arXiv:hep-ph*/0002160 (2000). *Get Preprint*

✱42 K. Aoki, D. Kusnezov. *arXiv:nlin.CD*/0103004 (2001). *Get Preprint*

✱43 T. Aoki. *Comput. Phys. Commun.* 142, 326 (2001). *DOI-Link*

✱44 H. Aoyama, J. Constable. *Literary Linguistic Comput.* 14, 339 (1999). *DOI-Link*

✱45 P. K. Aravind. *arXiv:quant-ph*/0210007 (2002). *Get Preprint*

✱46 C. Archer. *arXiv:quant-ph*/0312204 (2003). *Get Preprint*

✱47 P. Arena, S. Fazzino, L. Fortuna, P. Maniscalco. *Chaos, Solitons, Fractals* 17, 545 (2003). *DOI-Link*

✱48 T. A. Arias, T. D. Engeness. *arXiv:cond-mat*/9903259 (1999). *Get Preprint*

✱49 G. Arreaga, R. Capovilla, C. Chryssomalakos, J. Guven. *arXiv:cond-mat*/0103262 (2001). *Get Preprint*

✶50 J. J. Arulanandham, C. S. Calude, M. J. Dinneen. *Bull. Eur. Ass. Theor. Comput. Sc.* 76, 153 (2002).

✶51 J. J. Arulanandham in C. S. Calude, M. J. Dinneen, F. Peper (eds.). *Unconventional Models of Computation*, Springer-Verlag, Berlin, 2002. *BookLink (3)*

✶52 F. M. Arscott. *Periodic Differential Equations*, Pergamon Press, New York, 1964. *BookLink*

✶53 M. Aschbacher, A. M. Childs, P. Wocjan. *arXiv:quant-ph*/0412066 (2004). *Get Preprint*

✶54 R. Askey, D. T. Haimo. *Am. Math. Monthly* 103, 297 (1996).

✶55 R. Askey. *CRM Proc. Lecture Notes* 9, 13 (1997).

✶56 K. T. Atanassov. *Bull. Number Th.* 9, 18 (1985).

✶57 J. Auer, E. Krotscheck. *arXiv:cond-mat*/9811178 (1998). *Get Preprint*

✶58 J. Avery in J. L. Calais, E. S. Kryachko (eds.). *Structure and Dynamics of Atoms and Molecules: Conceptual Trends*, Kluwer, Dordrecht, 1995. *BookLink*

✶59 D. Avis, H. Imai, T. Ito, Y. Sasaki. *arXiv:quant-ph*/0404014 (2004). *Get Preprint*

✶60 M. Azreg–Ainou. *Europhys. Lett.* 62, 459 (2003). *DOI-Link*

✶61 V. Bach, F. Klopp, H. Zenk. *ESI Preprint* 1121 (2002). ftp://ftp.esi.ac.at:/pub/Preprints/esi1121.ps

✶62 M. E. Bacon, G. Heald, M. James. *Am. J. Phys.* 69, 38 (2001). *DOI-Link*

✶63 D. Bacon, J. Kempe, D. A. Lidar, K. B. Whaley, D. P. Divincenzo. *arXiv:quant-ph*/0102140 (2001). *Get Preprint*

✶64 F. Baginski, Q. Chen, I. Waldman. *Appl. Math. Model.* 25, 953 (2001). *DOI-Link*

✶65 J. Baez, J. W. Barrett. *arXiv:gr-qc*/9903060 (1999). *Get Preprint*

✶66 B. Bagchi, S. Mallik, C. Quesne. *arXiv:quant-ph*/0102093 (2001). *Get Preprint*

✶67 L. Y. Bahar. *Int. J. Nonl. Mech.* 35, 613 (2001). *DOI-Link*

✶68 P. Bak, K. Chen, M. Paczuski. *Phys. Rev. Lett.* 86, 2475 (2001). *DOI-Link*

✶69 B. B. Baker, E. T. Copson. *The Mathematical Theory of Huygens' Principle*, Clarendon Press, Oxford, 1950. *BookLink*

✶70 D. H. Bailey, J. M. Borwein, P. B. Borwein, S. Plouffe. *Math. Intell.* 19, n1, 590 (1997).

✶71 D. H. Bailey and J. M. Borwein. *CECM Preprint 99-143* (1999). http://www.cecm.sfu.ca/ftp/pub/CECM/Preprints/Postscript/99:143-Bailey-Borwein.ps.gz

✶72 D. H. Bailey, J. M. Borwein in B. Engquist, W. Schmid (eds.). *Mathematics Unlimited—2001 and Beyond*, Springer-Verlag, Berlin, 2001. *BookLink*

✶73 F. Baldovin, M. Novello, S. E. Perez Bergliaffa, J. M. Salim. *arXiv:gr-qc*/0003075 (2000). *Get Preprint*

✶74 G. Balmino. *Celest. Mech. Dynam. Astron.* 60, 331 (1994).

✶75 J. R. Banavar, A. Maritan. *Rev. Mod. Phys.* 75, 23 (2003). *DOI-Link*

✶76 S. Bandyopadhyay, P. O. Boykin, V. Roychowdhury, F. Vatan. *Algorithmica* 34, 512 (2002). *DOI-Link*

✶77 A.-L. Barabási, R. Albert. *Science* 286, 509 (1999). *DOI-Link*

✶78 R. M. Baram, H. J. Herrmann. *arXiv:cond-mat*/0312345 (2003). *Get Preprint*

✶79 R. M. Baram, H. J. Herrmann, N. Rivier. *arXiv:cond-mat*/0312460 (2003). *Get Preprint*

✶80 C. Barceló, S. Liberati, M. Visser. *arXiv:gr-qc*/0011026 (2000). *Get Preprint*

✶81 P. Barham. *The Science of Cooking*, Springer-Verlag, New York, 2002. *BookLink*

✶82 F. Barra, B. Davidovitch, I. Procaccia. *arXiv:cond-mat*/0105608 (2001). *Get Preprint*

✶83 F. Barra, B. Davidovitch, A. Levermann, I. Procaccia. *Phys. Rev. Lett.* 87, 134501 (2001). *DOI-Link*

✶84 F. Barra, H. G. E. Hentschel, A. Levermann, I. Procaccia. *arXiv:cond-mat*/0110089 (2001). *Get Preprint*

✶85 F. Barra, B. Davidovitch, I. Procaccia. *Phys. Rev.* E 65, 046144 (2002). *DOI-Link*

✶86 I. Bars. *arXiv:hep-th*/9804028 (1998). *Get Preprint*

✶87 W. Barthel, A. K. Hartmann, M. Leone, F. Ricci–Tersenghi, M. Weigt, R. Zecchina. *arXiv:cond-mat*/0111153 (2001). *Get Preprint*

✶88 T. Bartsch. *arXiv:physics*/0301017 (2003). *Get Preprint*

✶89 M. Barysz. *J. Chem. Phys.* 114, 9315 (2001). *DOI-Link*

✶90 M. Barysz, A. J. Sadlej. *J. Chem. Phys.* 116, 2696 (2002). *DOI-Link*

✶91 J. D. Bashford, I. Tsohantjis, P. D. Jarvis. *Proc. Natl. Acad. Sci. USA* 95, 987 (1998).

✶92 J. D. Bashford, P. D. Jarvis. *arXiv:physics*/0001066 (2000). *Get Preprint*

✶93 P. Bassanini, V. Bulgarelli. *Bollettino U.M.I.* 7, 8-A, 141 (1994).

✶94 A. Basu, R. S. Saraswat, K. B. Khare, G. P. Sastry, S. Bose. *Eur. J. Phys.* 23, 295 (2002). *DOI-Link*

✶95 H. Batelaan, T. J. Gay, J. J. Schwendiman. *Phys. Rev. Lett.* 79, 4517 (1997). *DOI-Link*

✶96 R. A. Battye, P. M. Sutcliffe. *Phys. Rev. Lett.* 81, 4798 (1998). *DOI-Link*

✶97 J. Baugh, D. R. Finkelstein, A. Galiautdinov, M. Shir–Garakani. *arXiv:hep-th*/0204031 (2003). *Get Preprint*

✶98 C. Baxa. *Math. Slovaca* 50, 531 (2000).

✶99 E. Baylis. *Theoretical Methods in the Physical Sciences*, Birkhäuser, Boston, 1994. *BookLink*

✶100 C. Beck, F. Schlögl. *Thermodynamics of Chaotic Systems*, Cambridge University Press, Cambridge, 1993. *BookLink (2)*

✶101 M. A. Bedau, J. S. McCaskill, N. H. Packard, S. Rasmussen, C. Adami, D. G. Green, T. Ikegami, K. Kaneko, T. S. Ray. *Artif. Life* 6, 363 (2001). *DOI-Link*

✶102 F. Behrooz. *Eur. J. Phys.* 18, 15 (1997). *DOI-Link*

✶103 F. Behroozi, H. K. Macomber, J. A. Dostal, C. H. Behroozi, B. K. Lambert. *Am. J. Phys.* 64, 1120 (1996). *DOI-Link*

✶104 J. D. Bekenstein. *arXiv:quant-ph*/0110005 (2001). *Get Preprint*

✶105 M. Belger, R. Schimming, V. Wünsch. *J. Anal. Appl.* 16, 9 (1997).

✶106 A. Belmonte, M. J. Shelley, S. T. Eldakar, C. H. Wiggins. *Phys. Rev. Lett.* 87, 114301 (2001). *DOI-Link*

✶107 E. G. Beltrametti, M. J. Maczynski. *J. Math.* 34, 4919 (1993).

✶108 C. M. Bender, S. Boettcher, L. R. Mead. *J. Math. Phys.* 35, 368 (1994). *DOI-Link*

✶109 C. M. Bender, S. Boettcher, M. Moshe. *J. Math. Phys.* 35, 4941 (1994). *DOI-Link*

✶110 C. Bender, S. Boettcher. *arXiv:physics*/9712001 (1997). *Get Preprint*

✶111 C. Bender, S. Boettcher. *arXiv:physics*/9801007 (1998). *Get Preprint*

✶112 C. M. Bender, G. V. Dunne, P. N. Meisinger. *arXiv:cond-mat*/9810369 (1998). *Get Preprint*

✶113 C. M. Bender, S. Boettcher, P. N. Meisinger. *arXiv:quant-ph*/9809072 (1998). *Get Preprint*

✶114 C. M. Bender, G. V. Dunne. *arXiv:quant-ph*/9812039 (1998). *Get Preprint*

✶115 C. M. Bender, G. V. Dunne, P. N. Meisinger. *Phys. Lett.* A 252, 272 (1999). *DOI-Link*

✶116 C. M. Bender, S. Boettcher, H. J. Jones, V. M. Savage. *arXiv:quant-ph*/9906057 (1999). *Get Preprint*

✶117 C. M. Bender, F. Cooper, P. N. Meisinger, V. M. Singer. *arXiv:quant-ph*/9907008 (1999). *Get Preprint*

✶118 C. M. Bender, S. Boettcher, V. M. Savage. *J. Math. Phys.* 41, 6381 (2000). *DOI-Link*

✶119 C. M. Bender. *Phys. Rep.* 315, 27 (1999). *DOI-Link*

✶120 C. M. Bender, G. V. Dunne, P. N. Meisinger, M. Simsek. *arXiv:quant-ph*/0101095 (2001). *Get Preprint*

✶121 I. Bengtson. *arXiv:quant-ph*/0406174 (2004). *Get Preprint*

✶122 I. Bengtsson, Å. Ericsson. *arXiv:quant-ph*/0410120 (2004). *Get Preprint*

✶123 G. Benenti, L. Galgani, J. -M. Strelcyn. *Phys. Rev.* A 14, 2338 (1976). *DOI-Link*

✶124 S. Benenti, C. Chanu, G. Rastelli. *J. Math. Phys.* 43, 5183 (2002). *DOI-Link*

✶125 C. A. Berenstein, A. Sebbar. *Adv. Math.* 110, 47 (1995). *DOI-Link*

✶126 L. Berg, M. Krüppel. *Z. Anal. Anw.* 19, 227 (2000).

✶127 L. Berg, M. Krüppel. *Result Math.* 38, 18 (2000).

✶128 M. Berger, P. Pansu, J.-P. Berry, X. Saint-Raymond. *Problems in Geometry*, Springer-Verlag, New York, 1984. *BookLink*

✶129 L. M. Berkovich. *Math. Comput. Simul.* 57, 175 (2001). *DOI-Link*

✶130 F. Bernardini. *Comput. Aided Design* 23, 51 (1991).

✶131 B. C. Berndt. *Ramanujan's Notebooks* v.1, Springer-Verlag, New York, 1985. *BookLink*

✶132 M. V. Berry. *Proc. R. Soc. Lond.* A 422, 7 (1989).

✶133 M. V. Berry. *J. Phys.* A 27, L391 (1994). *DOI-Link*

✶134 M. V. Berry in J. S. Anandan, J. L. Safko (eds.). *Proc. Int. Conf. Fundam. Aspects Quantum Theory*, World Scientific, Singapore, 1995.

✶135 M. V. Berry. *Proc. R. Soc. Lond.* A 452, 1207 (1996).

✶136 M. V. Berry, A. K. Geim. *Eur. J. Phys.* 18, 307 (1997). *DOI-Link*

✶137 M. V. Berry. *Found. Phys.* 31, 659 (2000). *DOI-Link*

✶138 M. J. Bertin, M. Pathiaux-Delefosse. *Conjecture de Lehmer et petits nombres de Salem, Queen's Papers in Pure and Applied Mathematics*, Kingston, 1989.

✶139 M. J. Bertin, A. Decomps-Guilloux, M. Grandet-Hugot, M. Pathiaux-Delefosse, J. P. Schreiber. *Pisot and Salem Numbers*, Birkhäuser, Basel, 1992. *BookLink*

✶140 U. Betke, M. Henk. *arXiv:math.MG*/9909172 (1999). *Get Preprint*

✶141 C. Betts. *J. Visual. Comput. Anim.* 9, 17 (1998).

✶142 A. T. Bharucha-Reid, M. Sambanham. *Random Polynomials*, Academic Press, Orlando, 1986. *BookLink (2)*

✶143 I. Bialynicki–Birula in E. Wolf (eds.). *Progress in Optics XXXVI*, Elsevier, Amsterdam, 1996.

✶144 G. Bianconi, A.-L. Barabási. *arXiv:cond-mat*/0011224 (2000). *Get Preprint*

✶145 T. Biedl, E. Demaine, M. Demaine, S. Lazard, A. Lubiw, J. O'Rourke, M. Overmars, S. Robbins, I. Streinu, G. Toussaint, S. Whitesides. *arXiv:cs.CG*/9910009 (1999). *Get Preprint*

✶146 L. Bildsten. *Phys. Rev.* E 66, 056309 (2002). *DOI-Link*

✶147 L. J. Billera, S. P. Holmes, K. Vogtman. *Adv. Appl. Math.* 27, 733 (2001). *DOI-Link*

✱148 K. Binsted, G. Ritchie. *Humor* 10, 25 (1997).

✱149 K. Binsted, G. Ritchie. *Humor* 14, 275 (2001).

✱150 D. Biswas. *arXiv:nlin.CD*/0107024 (2001). *Get Preprint*

✱151 D. Biswas. *arXiv:nlin.CD*/0107025 (2001). *Get Preprint*

✱152 N. Blachman, M. Mossinghoff. *Maple V Quick Reference*, Brooks/Cole, New York, 1994. *BookLink*

✱153 M. Blank. *arXiv:nlin.CD*/0003046 (2000). *Get Preprint*

✱154 C. Blatter. *Elem. Math.* 49, 155 (1994).

✱155 J. L. Blechschmidt, J. J. Uicker, Jr. *J. Mechanism, Transmissions, Automation Design* 108, 543 (1986).

✱156 A. M. Bloch, P. S. Krishnaprasad, J. E. Marsden, R. M. Murray. *Caltech CDS Reports* CIT/CDS 94-013 (1994). ftp://ftp.cds.caltech.edu/pub/cds/techreports/cds94-013.txt

✱157 A. M. Bloch. *Nonholonomic Mechanics and Control*, Springer-Verlag, New York, 2003. *BookLink*

✱158 A. M. Bloch, J. E. Marsden, D. V. Zenkov. *Notices Am. Math. Soc.* 52, 324 (2005).

✱159 H. Blok, H. A. Ferweda, H. K. Kuiken (eds.). *Huygens*' *Principle 1690-1990 Theory and Applications*, North Holland, Amsterdam, 1992. *BookLink*

✱160 F. J. Bloore, H. R. Morton. *Am. Math. Monthly* 92, 309 (1985).

✱161 M. L. Boas. *Am. J. Phys.* 29, 283 (1961).

✱162 R. P. Boas, Jr, H. Pollard. *Am. Math. Monthly* 80, 18 (1973).

✱163 R. P. Boas. *Am. Math. Monthly* 93, 644 (1986).

✱164 F. P. Boca, A. Zaharescu. *arXiv:math.NT*/0301270 (2003). *Get Preprint*

✱165 L. Bocquet. *arXiv:physics*/0210015 (2002). *Get Preprint*

✱166 E. Bogomolny. *arXiv:chao-dyn*/9910036 (1999). *Get Preprint*

✱167 C. S. Bohun, F. I. Cooperstock. *Phys. Rev.* A 60, 4291 (1999). *DOI-Link*

✱168 P. Boldi, M. Santini, S. Vigna. *Theor. Comput. Sc.* 282, 259 (2002). *DOI-Link*

✱169 H. Bondi. *Eur. J. Phys.* 14, 136 (1993). *DOI-Link*

✱170 J. Borgert, H. Schwarze. *Maple in der Physik*, Addison-Wesley, Bonn, 1995. *BookLink*

✱171 A. V. Borisov, I. S. Mamaev. *arXiv:nlin.SI*/0306002 (2003). *Get Preprint*

✱172 A. V. Borisov, I. S. Mamaev, A. A. Kilin. *Reg. Chaotic Dynam.* 8, 201 (2003). *DOI-Link*

✱173 F. Bornemann, D. Laurie, S. Wagon, J. Waldvogel. *The SIAM 100-Digit Challenge*, SIAM, Philadelphia, 2004.

BookLink

✱174 M. Borkovec, W. de Paris, R. Peikert. *Fractals* 2, 521 (1994).

✱175 P. B. Borwein. *J. Algor.* 6, 376 (1985).

✱176 J. Borwein, P. Borwein, R. Girgensohn, S. Parnes. *CECM Preprint* 032/95 (1995).
http://www.cecm.sfu.ca/ftp/pub/CECM/Preprints/Postscript/95:032-Borwein-Borwein-Girgensohn-Parnes.ps.gz

✱177 J. M. Borwein, R. M. Corless. *Am. Math. Monthly* 106, 889 (1999).

✱178 J. M. Borwein, P. B. Borwein. *CECM Preprint* 160/01 (2001).
http://www.cecm.sfu.ca/ftp/pub/CECM/Preprints/Postscript/01:160-Borwein-Borwein.ps.gz

✱179 P. Borwein. *Computational Excursions in Analysis and Number Theory*, Springer-Verlag, New York, 2002. *BookLink*

✱180 J. H. Borwein, D. H. Bailey. *Experimentation in Mathematics: Computational Paths to Discovery*, A K Peters, Dordrecht, Wellesley, 2003. *BookLink*

✱181 J. Borwein, D. Bailey. *Mathematics by Experiment*, A K Peters, Nautick, 2004. *BookLink*

✱182 J. A. Both, D. C. Hong, D. A. Kurtze. *Physica* A 301, 545 (2001). *DOI-Link*

✱183 N. M. Bou-Rabee, J. E. Marsden, L. A. Romero. *SIAM J. Appl. Dynam. Syst.* 3, 352 (2004). *DOI-Link*

✱184 D. W. Boyd. *Math. Comput.* 39, 249 (1982).

✱185 D. W. Boyd. *J. Number Th.* 21, 17 (1985). *DOI-Link*

✱186 J. N. Boyd, P. N. Raychowdhury. *Eur. J. Phys.* 17, 60 (1996). *DOI-Link*

✱187 J. P. Boyd. *Acta Appl. Math.* 56, 1 (1999). *DOI-Link*

✱188 T. H. Boyer. *Am. J. Phys.* 67, 954 (1999). *DOI-Link*

✱189 B. L. J. Braaksma, G. K. Immink, M. van der Put. *The Stokes Phenomena and Hilbert's 16th Problem*, World Scientific, Singapore, 1996. *BookLink*

✱190 M. Brack, R. J. Bhaduri. *Semiclassical Physics*, Addison-Wesley, Reading, 1997. *BookLink*

✱191 A. J. Bracken. *arXiv:quant-ph*/0210164 (2002). *Get Preprint*

✱192 A. J. Bracken, J. G. Wood. *Europhys. Lett.* 68, 1 (2004). *DOI-Link*

✱193 J. Brauer. *Finite Elem. Anal. Design* 40, 1857 (2004).

✱194 S. Braun. *Math. Comput. Simul.* 53, 249 (2000). *DOI-Link*

✱195 A. Braunstein, M. Leone, F. Ricci-Tersenghi, R. Zecchina. *J. Phys.* A 35, 7559 (2002). *DOI-Link*

✱196 R. P. Brent, E. M. McMillan. *Math. Comput.* 34, 305 (1980).

✶197 K. S. Brentner, F. Farassat. *J. Sound Vibr.* 170, 79 (1994). *DOI-Link*

✶198 L. Brewin. *arXiv:physics*/0307145 (2003). *Get Preprint*

✶199 R. Brito, M. J. Renne, C. van den Broeck. *Europhys. Lett.* A 350, 189 (2005). *DOI-Link*

✶200 P. Broadbridge, G. R. Fulford, N. D. Fowkes, D. Y. C. Chan, C. Lassig. *SIAM Rev.* 41, 363 (1999). *DOI-Link*

✶201 R. A. Broglia, G. Tiana. *arXiv:cond-mat*/0003096 (2000). *Get Preprint*

✶202 H. Brown, R. Bülow, J. Neubüser, H. Wondratschek, H. Zassenhaus. *Crystallographic Groups of Four-Dimen sional Space*, Wiley, New York, 1978. *BookLink*

✶203 E. Brown, H. Rabitz. *J. Math. Chem.* 31, 17 (2002). *DOI-Link*

✶204 I. Bruce. *Am. J. Phys.* 52, 1102 (1984). *DOI-Link*

✶205 T. A. Brun. *arXiv:gr-qc*/0209061 (2002). *Get Preprint*

✶206 A. D. Bruno. *Russ. Math. Surv.* 59, 429 (2004). *DOI-Link*

✶207 D. Bruß. *J. Math. Phys.* 43, 4237 (2002). *DOI-Link*

✶208 T. N. Buch, W. B. Pardo, J. A. Walkenstein, M. Monti, E. Rosa, Jr. *Phys. Lett.* A 248, 353 (1998). *DOI-Link*

✶209 B. Buchberger. *SIGSAM Bull.* 36, 3 (2002). *DOI-Link*

✶210 J. Buhler, D. Eisenbud, R. Graham, C. Wright. *Am. Math. Monthly* 101, 507 (1994).

✶211 F. Bullo, A. D. Lewis. *Geometric Control of Mechanical Systems*, Springer, New York, 2005. *BookLink*

✶212 R. Bülow, J. Neubüser, H. Wondratschek. *Acta Cryst.* A 27, 520 (1971).

✶213 P. L. Buono, M. Golubitsky. *J. Math. Biol.* 42, 291 (2001). *DOI-Link*

✶214 C. Burdík, O. Navrátil. *arXiv:nlin.SI*/0008019 (2000). *Get Preprint*

✶215 Č. Burdík, O. Navrátil. *J. Phys.* A 35, 2431 (2002). *DOI-Link*

✶216 F. Burgbacher, C. Lämmerzahl, A. Macias. *J. Math. Phys.* 40, 625 (1999). *DOI-Link*

✶217 R. Burridge, L. Knopoff. *Bull. Seis. Soc. Am.* 57, 341 (1967).

✶218 C. Burstedde, K. Klauck, A. Schadschneider, J. Zittartz. *arXiv:cond-mat*/0102397 (2001). *Get Preprint*

✶219 C. Burstedde, A. Kirchner, K. Klauck, A. Schadschneider, J. Zittartz. *arXiv:cond-mat*/0112119 (2001). *Get Preprint*

✶220 L. G. Bushnel, D. Tilbury, S. S. Sastry. *Int. J. Robot. Res.* 14, 366 (1995).

✱221 V. P. Bykov, A. A. Zadernovskii. *Opt. Spektrosc.* 48, 130 (1980).

✱222 E. Caglioti, N. Chernov, J. L. Lebowitz. *arXiv:cond-mat*/0302345 (2003). *Get Preprint*

✱223 E. R. Caianiello. *Nuov. Cim.* S 14, 177 (1959).

✱224 G. Caldarelli. *arXiv:cond-mat*/0011086 (2000). *Get Preprint*

✱225 M. S. Calder, A. Kempf. *arXiv:quant-ph*/0405065 (2004). *Get Preprint*

✱226 C. S. Calude, T. Zamfirescu. *New Zealand J. Math.* 27, 7 (1998).

✱227 C. S. Calude, B. Pavlov. *Quant. Inform. Process.* 1, 107 (2002). *DOI-Link*

✱228 L. M. B. C. Campos. *IMA J. Appl. Math.* 33, 109 (1984).

✱229 F. Cannata, G. Junker, J. Trost. *arXiv:quant-ph*/9805085 (1998). *Get Preprint*

✱230 F. Cannata, G. Junker, J. Trost. *Phys. Lett.* A 246, 219 (1998). *DOI-Link*

✱231 J. Cantarella, D. De Turck, M. Teytel. *J. Math. Phys.* 41, 5615 (2000). *DOI-Link*

✱232 M. Cantor. *Vorlesungen zur Geschichte der Mathematik*, Teubner, Leipzig, 1913. *BookLink(4)*

✱233 R. Capovilla, C. Chryssomalakos, J. Guven. *J. Phys.* A 35, 6571 (2002). *DOI-Link*

✱234 J. L. Cardoso, R. Álvarzez–Nodarse. *J. Phys.* A 36, 2055 (2003). *DOI-Link*

✱235 J. F. Cariñena, G. Marmo, J. Nasarre. *Int. J. Mod. Phys.* 13, 3601 (1998). *DOI-Link*

✱236 J. F. Cariñena, J. Grabowski, G. Marmo. *Rep. Math. Phys.* 48, 47 (2001). *DOI-Link*

✱237 J. F. Cariñena, J. Grabowski, A. Ramos. *Acta Appl. Math.* 66, 67 (2001). *DOI-Link*

✱238 J. P. Carini, J. T. Londergan, K. Mullen, D. P. Murdock. *Phys. Rev.* B 46, 15538 (1992). *DOI-Link*

✱239 J. P. Carini, J. T. Londergan, K. Mullen, D. P. Murdock. *Phys. Rev.* B 48, 4503 (1993). *DOI-Link*

✱240 J. P. Carini, J. T. Londergan, D. P. Murdock. *Phys. Rev.* B 55, 9852 (1997). *DOI-Link*

✱241 M. Carley. *J. Sound Vibr.* 244, 1 (2001). *DOI-Link*

✱242 M. C. Casdagli. *Physica* D 108, 12 (1997). *DOI-Link*

✱243 L. W. Casperson. *J. Sound Vibr.* 162, 251 (1993). *DOI-Link*

✱244 L. W. Casperson. *J. Appl. Phys.* 74, 4894 (1993). *DOI-Link*

✱245 G. Castellani, S. Sivasubramanian, A. Widom, Y. N. Srivastava. *arXiv:quant-ph*/0306185 (2003). *Get Preprint*

✱246 C. Castro. *arXiv:physics*/0101104 (2001). *Get Preprint*

✱247 O. A. Cahlykh, M. V. Feigin, A. P. Vesslov. *arXiv:math-ph*/9903019 (1999). *Get Preprint*

✱248 N. Calkin, H. S. Wilf. *Am. Math. Monthly* 107, 360 (2000).

✱249 F. Calogero. *Variable Phase Shift Approach to Potential Scattering*, Academic Press, New York, 1967. *BookLink*

✱250 C. S. Calude, E. Calude, S. Marcus. *arXiv:math.HO*/0305123 (2003). *Get Preprint*

✱251 M. A. Can, O. Cakir, A. Horzela, E. Kapuscik, A. A. Klyachko, A. S. Shumovsky. *arXiv:quant-ph*/0308093 (2003). *Get Preprint*

✱252 Y. Cao, Z. Hou, Y. Liu. *Phys. Lett.* A 327, 247 (2004). *DOI-Link*

✱253 A. Cayley. *Proc. Lond. Math. Soc.* 6, 20 (1874).

✱254 A. Celletti in . D. Benest, C. Froeschlé (eds.). *Singularities in Gravitational Systems*, Springer-Verlag, Berlin, 2002. *BookLink*

✱255 H. Cendra, J. E. Marsden, T. S. Ratiu in B. Engquist, W. Schmid (eds.). *Mathematics Unlimited—2001 and Beyond*, Springer-Verlag, Berlin, 2001. *BookLink*

✱256 E. Cerda, L. Mahadevan. *Phys. Rev. Lett.* 90, 074302 (2003). *DOI-Link*

✱257 J. L. Cereceda. *arXiv:quant-ph*/0003026 (2000). *Get Preprint*

✱258 K. Chadan, R. Kobayashi, T. Kobayashi. *J. Math. Phys.* 42, 4031 (2001). *DOI-Link*

✱259 C. J. Chaitin. *The Unknowable* (1999). http://www.cs.auckland.ac.nz/CDMTCS/chaitin/unknowable/

✱260 A. Chakraborti, B. K. Chakraborti. *arXiv:cond-mat*/0002022 (2000). *Get Preprint*

✱261 F. Chamizo, A. Córdoba. *Adv. Math.* 142, 335 (1999). *DOI-Link*

✱262 A. R. Champneys, W. B. Fraser. *Proc. R. Soc. Lond.* A 456, 553 (2000). *DOI-Link*

✱263 G. K.-L. Chan, P. W. Ayers, E. S. Croot, III. *J. Stat. Phys.* 109, 289 (2002). *DOI-Link*

✱264 K. Chan, C. Law, J. Eberly. *Phys. Rev. Lett.* 88, 100402 (2002). *DOI-Link*

✱265 C.-H. Chang, T. Y. Tsong. *Phys. Rev.* E 67, 025101 (2003). *DOI-Link*

✱266 B. W. Char, K. O. Geddes, G. H. Gonnet, B. L. Leong, M. B. Monagan, S. M. Watt. *Maple V Language Refer‐ ence Manual*, Springer-Verlag, New York, 1991. *BookLink*

✱267 B. W. Char, K. O. Geddes, G. H. Gonnet, B. L. Leong, M. B. Monagan, S. M. Watt. *Maple V Library Reference Manual*, Springer-Verlag, New York, 1991. *BookLink*

✱268 B. W. Char, K. O. Geddes, G. H. Gonnet, B. L. Leong, M. B. Monagan, S. M. Watt. *First Leaves, A Tutorial Introduction to Maple*, Springer-Verlag, New York, 1991. *BookLink (2)*

✱269 S. Chaturvedi. *Phys. Rev.* A 65, 044301 (2002). *DOI-Link*

✱270 L. Y. Chen, N. Goldenfeld, Y. Ono. *Phys. Rev.* E 51, 5577 (1996). *DOI-Link*

✱271 Y. Chen, Z. Yan, H. Zhang. *Appl. Math. Mech.* 24, 256 (2003).

✱272 R. Cheng, A. Dasgupta, B. R. Ebanks, L. F. Kinch, L. M. Larson. R. B. McFadden. *Am. Math. Monthly* 105, 704 (1998).

✱273 T. Cheon. *Phys. Lett.* A 248, 285 (1998). *DOI-Link*

✱274 T. Cheon, T. Fülöp, I. Tsutsui. *arXiv:quant-ph*/0008123 (2000). *Get Preprint*

✱275 T. Cheon. *arXiv:quant-ph*/0203041 (2002). *Get Preprint*

✱276 N. Chernov, J. L. Lebowitz. *mp_arc* 02-223 (2002). http://rene.ma.utexas.edu/mp_arc/c/02/02-223.ps.gz

✱277 N. Chernov, J. L. Lebowitz, Y. Sinai. *J. Stat. Phys.* 109, 529 (2002). *DOI-Link*

✱278 N. Chernov, J. L. Lebowitz, Y. Sinai. *arXiv:cond-mat*/0301163 (2003). *Get Preprint*

✱279 N. Chernov. *arXiv:cond-mat*/0303395 (2003). *Get Preprint*

✱280 M.-D. Choi. *Am. Math. Monthly* 90, 301 (1983).

✱281 T. Y. Chow. *arXiv:math.NT*/9805045 (1998). *Get Preprint*

✱282 T. Y Chow. *arXiv:math.LO*/9903160 (1999). *Get Preprint*

✱283 D. Chowdhury, L. Santen, A. Schadschneider. *Phys. Rep.* 329, 199 (2000). *DOI-Link*

✱284 D. Chruscinski. *arXiv:math-ph*/0206009 (2002). *Get Preprint*

✱285 E. T. Chung, J. Zou. *SIAM J. Matrix Anal.* 24, 1149 (2003). *DOI-Link*

✱286 M.-C. Chung, I. Peschel. *Phys. Rev.* B 64, 064412 (2001). *DOI-Link*

✱287 R. Cianco, A. Khrennikov. *Int. J. Theor. Phys.* 33, 1217 (1994). *DOI-Link*

✱288 B. Cipra, P. Zorn (ed.). *What's Happening in the Mathematical Sciences 1998-1999*, American Mathematical Society, Providence, 1999. *BookLink*

✱289 M. A. Cirone, J. P. Dahl, M. Fedorov, D. Greenberger, W. P. Schleich. *arXiv:quant-ph*/0108083 (2001). *Get Preprint*

✱290 R. Cocan, J. O'Rourke. *arXiv:cs.CG*/9908005 (1999). *Get Preprint*

✱291 S. Cocco, R. Monasson. *Phys. Rev.* E 66, 037101 (2002). *DOI-Link*

✱292 M. W. Coffey. *Phys. Lett.* A 304, 8 (2002). *DOI-Link*

✱293 R. J. Cohen. *Am. J. Phys.* 45, 12 (1977). *DOI-Link*

✱294 H. Cohen in M. Waldschmidt, P. Moussa, J.-M. Luck, C. Itzykson (eds.). *From Number Theory to Physics*, Springer-Verlag, Berlin, 1992. *BookLink*

✶295 A. M. Cohen, J. H. Davenport, A. J. P. Heck in A. M. Cohen (ed.). *Computer Algebra for Industry—Problem Solving in Practice: A Survey of Applications and Techniques*, Wiley, Chichester, 1993. *BookLink*

✶296 M. J. Coleman, A. Ruina. *Phys. Rev. Lett.* 80, 3658 (1998). *DOI-Link*

✶297 M. J. Coleman, M. Garcia, K. Mombaur, A. Ruina. *arXiv:physics*/0104034 (2001). *Get Preprint*

✶298 J. J. Collins, I. N. Stewart. *Biol. Cybern.* 68, 287 (1993).

✶299 J. J. Collins, I. N. Stewart. *J. Nonlin. Sci.* 3, 349 (1993).

✶300 L. Colzani, M. Vignati. *J. Approx. Th.* 80, 119 (1995). *DOI-Link*

✶301 *Computeralgebra in Deutschland Bestandsaufnahme, Möglichkeiten, Perspektiven*, published by Fachgruppe of the GI, DMV, GAMM, Passau and Heidelberg, http://www.uni-karlsruhe.de/~CAIS/mitteilungen/ca-report-info.ps, 1993.

✶302 J. Constable, H. Aoyama. *Literary Linguistic Comput.* 14, 507 (1999). *DOI-Link*

✶303 J. Constable, H. Aoyama. *arXiv:cs.CL*/0109039 (2001). *Get Preprint*

✶304 P. C. Consul. *Int. Stat. Rev.* 59, 271 (1991).

✶305 B. Cordani. *J. Phys.* A 22, 2695 (1989). *DOI-Link*

✶306 R. M. Corless. *Essential Maple*, Springer-Verlag, New York, 1994. *BookLink (2)*

✶307 R. M. Corless. *Symbolic Recipes*, Springer-Verlag, New York, 1994. *BookLink*

✶308 R. Corless. *SIGSAM Bull.* 31, n3, 1 (1997).

✶309 F. H. J. Cornish. *J. Phys.* A 17, 323 (1984). *DOI-Link*

✶310 J. Cortés, M. de León, D. M. de Diego. *arXiv:math.DG*/0006183 (2000). *Get Preprint*

✶311 J. Cortés Monforte. *Geometric, Control and Numerical Aspects of Nonholonomic Systems*, Springer-Verlag, Berlin, 2002. *BookLink*

✶312 U. M. S. Costa, M. L. Lyra. *Phys. Rev.* E 56, 245 (1997). *DOI-Link*

✶313 A. A. Cottey. *Am. J. Phys.* 39, 1235 (1971).

✶314 K. Cottrill–Shepher, M. Naber. *J. Math. Phys.* 42, 2203 (2001). *DOI-Link*

✶315 P. Coullet, L. Mahadevan, C. Riera. *Progr. Theor. Phys.* Suppl. 139, 507 (2000).

✶316 P. Coullet, L. Mahadevan, C. S. Riera. *arXiv:physics*/0408096 (2004). *Get Preprint*

✶317 F. A. B. Coutinho, Y. Nogami, L. Tomio. *arXiv:quant-ph*/9903098 (1999). *Get Preprint*

✶318 D. R. Cox, F. R. S. Isham, V. Isham. *Proc. R. Soc. Lond.* A 415, 317 (1988).

✱319 H. S. M. Coxeter. *Am. Math. Monthly* 75, 5 (1968).

✱320 R. Cowan, A. K. L. Tsang. *Adv. Appl. Prob.* 26, 54 (1994).

✱321 T. Craig. *Am. J. Math.* 8, 85 (1885).

✱322 T. Craig. *Am. J. Math.* 9, 97 (1885).

✱323 M. Cremer. *Der Verkehrsfluß auf Schnellstraßen*, Springer-Verlag, Berlin, 1979. *BookLink*

✱324 T. Cremer. *Interpretationsprobleme der speziellen Relativitätstheorie*, Harri Deutsch, Frankfurt am Main, 1990. *BookLink*

✱325 H. T. Croft, K. J. Falconer, R. K. Guy. *Unsolved Problems in Geometry*, Springer-Verlag, New York, 1991. *BookLink*

✱326 G. E. Crooks. *Phys. Rev.* E 61, 2361 (2000). *DOI-Link*

✱327 B. Crosignani, P. Di Porto. *Europhys. Lett.* 53, 290, (2001). *DOI-Link*

✱328 B. Crosignani, P. Di Porto, C. Conti. *arXiv:physics*/0207073 (2002). *Get Preprint*

✱329 B. Crosignani, P. Di Porto, C. Conti in D. P. Sheehan (ed.). *Quantum Limits to the Second Law*, American Institute of Physics, New York, 2002. *BookLink*

✱330 B. Crosignani, P. Di Porto, C. Conti. *arXiv:phsyics*/0410050 (2004). *Get Preprint*

✱331 Z. Csahók, C. Misbah, F. Rioual, A. Valance. *arXiv:cond-mat*/0001336 (2000). *Get Preprint*

✱332 H. B. Curry, R. Feys, W. Craig. *Combinatory Logic*, v.1, North Holland, Amsterdam, 1958. *BookLink*

✱333 R. Cushman, J. Hermans, D. Kemppainen in H. W. Broer, S. A. van Gils, I. Hoveijn, F. Takens (eds.). *Nonlinear Dynamical Systems and Chaos*, Birkhäuser, Basel, 1996. *BookLink*

✱334 A. Czirók, T. Vicsek in D. Reguera, J. M. G. Vilar, J. M. Rubi (eds.). *Statistical Mechanics of Biocomplexity*, Springer-Verlag, Berlin, 1999. *BookLink*

✱335 J. Czyz. *Paradoxes of Measures and Dimensions Originating in Felix Hausdorff's Ideas*, World Scientific, Singapore, 1993. *BookLink*

✱336 A. B. O. Daalhuis. *Proc. R. Soc. Edinb.* A 123, 731 (1993).

✱337 A. B. O. Daalhuis in E. Koelink, W. Van Assche (eds.). *Orthogonal Polynomials and Special Functions*, Springer-Verlag, Berlin, 2003. *BookLink*

✱338 A. Daerr, P. Lee, J. Lanuza, É. Clement. *arXiv:cond-mat*/0205632 (2002). *Get Preprint*

✱339 S. Daftuar, P. Hayden. *arXiv:quant-ph*/0410052 (2004). *Get Preprint*

✱340 P. A. Dando, T. S. Monteiro. *arXiv:physics*/9803019 (1998). *Get Preprint*

✱341 J. M. Daniels. *Can. J. Phys.* 74, 236 (1996).

✱342 B. T. Darling. *Opt. Acta* 31, 97 (1984).

✱343 A. D' Avanzo, G. Marmo. *arXiv:math-ph*/0411014 (2004). *Get Preprint*

✱344 B. Davidovitch, H. G. E. Hentschel, Z. Olami, I. Procaccia, L. M. Sander, E. Somfai. *Phys. Rev.* E 59, 1368 (1999). *DOI-Link*

✱345 B. Davidovitch, M. J. Feigenbaum, H. G. E. Hentschel, I. Procaccia. *arXiv:cond-mat*/0002420 (2000). *Get Preprint*

✱346 P. J. Davis, P. Rabinowitz in F. L. Alt (ed.). *Advances in Computers*, Academic Press, New York, 1961. *BookLink*

✱347 M. A. B. Deakin. *Math. Mag.* 45, 246 (1972).

✱348 O. F. de Alcantara Bonfim, D. Griffith. *Am. J. Phys.* 69, 515 (2001). *DOI-Link*

✱349 T. A. de Alwis. *Coll. Math. J.* 26, 361 (1995).

✱350 A. De Angelis. *Eur. J. Phys.* 8, 201 (1987). *DOI-Link*

✱351 S. De Biévre. *mp_arc* 01-207 (2001). http://rene.ma.utexas.edu/mp_arc/c/01/01-207.ps.gz

✱352 B. DeFacio, C. L. Hammer. *J. Math. Phys.* 15, 1071 (1974). *DOI-Link*

✱353 W. L. F. Degen in M. Dählen, T. Lyche, L. L. Shuhmaker (eds.). *Mathematical Methods for Curves and Sur faces*, Vanderbilt University Press, Nashville, 1995. *BookLink (2)*

✱354 R. de la Llave, B. Buchberger, S. Matvev, M. Seppälä in C. Casacuberta, R. M. Miró–Roig, J. M. Ortega, S. Xambó–Descamps. *Mathematical Glimpses into the 21st Century*, Societat Catalana de Matematiques, Barcelona, 2001.

✱355 A. C. de la Torre, D. Goyeneche. *Am. J. Phys.* 71, 49 (2002). *DOI-Link*

✱356 A. C. de la Torre. *arXiv:quant-ph*/0410179 (2004). *Get Preprint*

✱357 R. Delbourgo. *Am. J. Phys.* 55, 799 (1987). *DOI-Link*

✱358 T. K. DeLillo, A. R. Elcrat, J. A. Pfaltzgraff. *SIAM Rev.* 43, 469 (2001). *DOI-Link*

✱359 P. De Los Rios, O. Pla. *Phys. Rev.* E 61, 5620 (2000). *DOI-Link*

✱360 A. De Luca, L. M. Ricciardi, and R. Vasudevan. *J. Math. Phys.* 11, 530 (1970). *DOI-Link*

✱361 E. D. Demaine, M. L. Demaine, A. Lubiw. *Proc. Japan. Conf. Discrete Comput. Geom.*, Springer-Verlag, Tokyo, 1998. *BookLink*

✱362 J. W. Demmel. *Math. Comput.* 50, 449 (1988).

✱363 H. H. Denman. *Am. J. Phys.* 53, 224 (1985). *DOI-Link*

✱364 M. Denny. *Can. J. Phys.* 76, 295 (1998). *DOI-Link*

✱365 M. Denny. *Can. J. Phys.* 77, 923 (2000). *DOI-Link*

✱366 C. R. de Oliveira, G. Q. Pellegrino. *J. Phys.* A 34, L239 (2001). *DOI-Link*

✱367 G. Derfel, R. N. Sen. *Open Sys. Inform. Dyn.* 4, 125 (1997). *DOI-Link*

✱368 B. Derrida, S. C. Manrubia, D. H. Zanette. *Phys. Rev. Lett.* 82, 1987 (1999). *DOI-Link*

✱369 B. Derrida, S. C. Manrubia, D. H. Zanette. *arXiv:cond-mat*/9912059 (1999). *Get Preprint*

✱370 B. Derrida, S. C. Manrubia, D. H. Zanette. *arXiv:physics*/0003016 (2000). *Get Preprint*

✱371 M. de Sousa Viera. *arXiv:cond-mat.*/9907201 (1999). *Get Preprint*

✱372 M. de Sousa Viera. *Phys. Rev. Lett.* 82, 201 (1999). *DOI-Link*

✱373 L. C. Detwiler, J. R. Klauder. *Phys. Rev.* D 11, 1436 (1975). *DOI-Link*

✱374 A. J. Devaney, E. Wolf. *Phys. Rev.* D 8, 1044 (1973). *DOI-Link*

✱375 J. S. Devitt. *Calculus with Maple V*, Brooks/Cole, Pacific Grove, 1993. *BookLink*

✱376 J. de Vries, S. Luther, D. Lohse. *Eur. J. Phys.* B 29, 503 (2002). *DOI-Link*

✱377 R. L. Dewar, S. R. Hudson. *Physica* D 112, 275 (1998). *DOI-Link*

✱378 A. Dhar. *Phys. Rev. Lett.* 86, 3554 (2001). *DOI-Link*

✱379 A. Dhar. *arXiv:cond-mat*/0210470 (2002). *Get Preprint*

✱380 A. Di Bucchianico, D. Loeb. *Electr. J. Combinatorics* DS3 (2000).
http://www.combinatorics.org/Surveys/ds3.pdf

✱381 R. Dickman. *arXiv:cond-mat*/0210327 (2002). *Get Preprint*

✱382 B. A. DiDonna, T. A. Witten, E. M. Kramer. *arXiv:math-ph*/0101002 (2001). *Get Preprint*

✱383 B. A. DiDonna. *Phys. Rev.* E 66, 016601 (2002). *DOI-Link*

✱384 B. Diggs, G. Genovese, J. B. Kadane, R. H. Swendson. *Comput. Phys. Commun.* 121/122, 1 (1999). *DOI-Link*

✱385 R. Ding, D. Schattschneider, T. Zamfirescu. *Discr. Math.* 221, 113 (2000). *DOI-Link*

✱386 L. Dinis, J. M. R. Parrondo. *arXiv:cond-mat*/0212358 (2002). *Get Preprint*

✱387 A. D. D'Innocenzo, F. Paladinii, L. Renna. *Phys. Rev.* E 65, 056208 (2002). *DOI-Link*

✱388 P. A. M. Dirac. *The Principles of Quantum Mechanics*, Oxford University Press, Oxford, 1930. *BookLink (2)*

✱389 M. M. Djrbashian. *Harmonic Analysis, and Boundary Value Problems in the Complex Domain*, Birkhäuser, Boston, 1993. *BookLink*

✱390 P. S. Dodds, D. H. Rothman. *arXiv:physics*/0005047 (2000). *Get Preprint*

✱391 P. S. Dodds, D. H. Rothman. *arXiv:physics*/0005048 (2000). *Get Preprint*

✱392 P. S. Dodds, D. H. Rothman. *arXiv:physics*/0005049 (2000). *Get Preprint*

✱393 P. S. Dodds, J. S. Weitz. *Phys. Rev.* E 65, 056108 (2002). *DOI-Link*

✱394 J. R. Dorfman. *An Introduction to Chaos in Nonequilibrium Statistical Mechanics*, Cambridge University Press, Cambridge, 1999. *BookLink*

✱395 J. D. Dollard, C. N. Friedman. *Product Integration with Applications to Differential Equations*, Addison-Wesley, Reading, 1979. *BookLink*

✱396 C. Domb in G. R. Grimmett, D. J. A. Welsh (eds.). *Disorder in Physical Systems*, Clarendon Press, Oxford, 1990. *BookLink*

✱397 C. C. Donato, M. A. F. Gomes, R. E. Souza. *Phys. Rev.* E 66, 015102(R) (2002). *DOI-Link*

✱398 M. A. Doncheski, R. W. Robinett. *Ann. Phys.* 299, 208 (1985). *DOI-Link*

✱399 P. S. Donelan, C. G. Gibson in B. Bruce, D. Mond (eds.). *Singularity Theory*, Cambridge University Press, Cambridge, 1999. *BookLink*

✱400 J. Dongarra, T. Rowan, R. Wade. *ACM Trans. Math. Softw.* 21, 79 (1995). *DOI-Link*

✱401 P. Dorey, C. Dunning, R. Tateo. *arXiv:hep-th*/0010148 (2000). *Get Preprint*

✱402 J. Dreitlein. *Found. Phys.* 23, 923 (1993).

✱403 R. M. Dreizler, E. K. U. Gross. *Density Functional Theory*, Springer-Verlag, Berlin, 1990. *BookLink*

✱404 D. A. Drew in W. E. Boyce (ed.). *Case Studies in Mathematical Modelling*, Pitman, Boston, 1981.
BookLink

✱405 D. A. Drew in M. Braun, C. S. Coleman, D. A. Drew (eds.). *Differential Equations Models*, Springer-Verlag, New York, 1983. *BookLink*

✱406 O. Dreyer, R. Puzio. *J. Math. Biol.* 43, 144 (2001). *DOI-Link*

✱407 M. Duarte, V. M. Zatsiorsky. *Phys. Lett.* A 283, 124 (2001). *DOI-Link*

✱408 A. Dubickas. *Publ. Math. Debrecen* 56, 141 (2000).

✱409 O. Dubois, Y. Boufkhad, J. Mandler. *arXiv:cs.DM*/0211036 (2002). *Get Preprint*

✱410 P. Duclos, P. Exner. *Rev. Math. Phys.* 7, 73 (1995).

✱411 P. Duclos, P. Exner, D. Krejcirik. *arXiv:quant-ph*/9910035 (1999). *Get Preprint*

✱412 M. J. Duff, L. B. Okun, G. Veneziano. *arXiv:physics*/0110060 (2001). *Get Preprint*

✱413 R. J. Duffin. *Proc. Am. Math. Soc.* 13, 965 (1963).

✱414 J. J. Duistermaat. *arXiv:math.DS*/0409019 (2004). *Get Preprint*

✱415 H. R. Dullin, R. W. Easton. *Physica* D 126, 1 (1999). *DOI-Link*

✱416 J. Dunkel, P. Hänggi. *arXiv:cond-mat*/0411011 (2004). *Get Preprint*

✱417 D. Duplij, S. Duplij. *arXiv:physics*/0006062 (2000). *Get Preprint*

✱418 O. Durán, V. Schwämmle, H. Herrmann. *arXiv:cond-mat*/0406392 (2004). *Get Preprint*

✱419 M. S. Dutra, A. C. de Pina Filho, V. F. Romano. *Biol. Cybern.* 88, 286 (2003). *DOI-Link*

✱420 C. Duval, P. A. Horváthy. *arXiv:hep-th*/0002233 (2000). *Get Preprint*

✱421 K. G. Dyall. *J. Comput. Chem.* 23, 786 (2002). *DOI-Link*

✱422 J. Earman. *Bangs, Crunches, Whimpers, and Shrieks*, Oxford University Press, New York, 1995. *BookLink*

✱423 M. S. P. Eastham. *The Theory of Periodic Differential Operators*, Scottish Academic Press, Edinburgh, 1973. *BookLink*

✱424 K. Easwar, F. Rouyer, N. Menon. *Phys. Rev.* E 66, 045102(R) (2002). *DOI-Link*

✱425 S. Ebenfeld, F. Scheck. *Ann. Phys.* 243, 195 (1995). *DOI-Link*

✱426 K. Eckert, J. Schliemann, D. Bruß, M. Lewenstein. *Ann. Phys.* 299, 88 (2002). *DOI-Link*

✱427 J.-P. Eckmann, S. O. Kamphorst, D. Ruelle. *Europhys. Lett.* 4, 973 (1987).

✱428 J.-P. Eckmann, E. Zabey. *J. Stat. Phys.* 114, 515 (2004). *DOI-Link*

✱429 J. Egger. *Physica* D 165, 127 (2002). *DOI-Link*

✱430 I. L. Egusquiza, M. A. Valle Basagoiti. *Phys. Rev.* A 57, 1586 (1998). *DOI-Link*

✱431 R. Ehrenborg, M. Readdy. *Discr. Math.* 157, 107 (1996). *DOI-Link*

✱432 R. Ehrlich. *Why Toast Lands Jelly-Down*, Princeton University Press, Princeton, 1997. *BookLink*

✱433 S.-I. Ei, K. Fujii, T. Kunihiro. *arXiv:hep-th*/9905088 (1999). *Get Preprint*

✱434 S.-I. Ei, K. Fujii. *Ann. Phys.* 280, 236 (2000). *DOI-Link*

✱435 S. N. Elaydi. *Discrete Chaos*, Chapman & Hall, Boca Raton, 2000. *BookLink*

✱436 A. Elci. *Ann. Phys.* 229, 221 (1994). *DOI-Link*

✱437 M. S. El Naschie. *Chaos, Solitons, Fractals* 14, 1117 (2002). *DOI-Link*

✱438 A. El-Sonbaty, H. Stachel in A. Wyzkowski, T. Dyduch, R. Gorska, L. Piekarski, L. Zakowska (eds.). *Proceed* *ings of the 7th International Conference on Engineering Computer Graphics and Descriptive Geometry*, Cracow, 1996. *BookLink*

✱439 G. S. Ely. *Am. J. Math.* 5, 337 (1882).

✱440 P. Enders. *Eur. J. Phys.* 17, 226 (1996). *DOI-Link*

✱441 B.-G. Englert, Y. Aharonov. *arXiv:quant-ph*/0101134 (2001). *Get Preprint*

✱442 A. Enneper. *Elliptische Functionen*, Louis Nebert, Halle, 1890.

✱443 R.H. Enns, G. McGuire. *Nonlinear Physics with Maple for Scientists*, Birkhäuser, Basel, 1997. *BookLink (3)*

✱444 R. Enns, G. McGuire. *Computer Algebra Recipes*, Birkhäuser, Boston, 2002. *BookLink (2)*

✱445 D. Eppstein. *arXiv:cs.CG*/0106032 (2001). *Get Preprint*

✱446 P. Erdös, G. Schibler, R. C. Herndorn. *Am. J. Phys.* 60, 335 (1992). *DOI-Link*

✱447 G. Etesi, I. Németi. *Int. J. Theor. Phys.* 41, 341 (2002). *DOI-Link*

✱448 D. J. Evans, E. G. D. Cohen, G. P. Morris. *Phys. Rev. Lett.* 71, 2401 (1993). *DOI-Link*

✱449 D. J. Evans, D. J. Searles. *Adv. Phys.* 51, 1529 (2002). *DOI-Link*

✱450 P. Exner, E. M. Harrell, M. Loss. *arXiv:math-ph*/9901022 (1999). *Get Preprint*

✱451 P. Exner, H. Grosse. *arXiv:math-ph*/9910029 (1999). *Get Preprint*

✱452 J. Fajans. *Am. J. Phys.* 68, 654 (2000). *DOI-Link*

✱453 M. Falcioni, L. Palatella, S. Pigolotti, A. Vulpiani. *arXiv:nlin.CD*/0503035 (2005). *Get Preprint*

✱454 K. Farahmand, A. Shaposhnikov. *Appl. Math. Lett.* 17, 1085 (2004). *DOI-Link*

✱455 F. Farassat, K. S. Brentner. *Theor. Comput. Fluid Dyn.* 10, 155 (1998). *DOI-Link*

✱456 F. Farassat. *J. Sound Vibr.* 239, 785 (2001). *DOI-Link*

✱457 Z. Farkas, G. Bartels, T. Unger, D. E. Wolf. *arXiv:physics*/0210024 (2002). *Get Preprint*

✱458 R. T. Farouki in P.-J. Laurent, P. Sablonniere, L. L. Schumaker, (eds.). *Curve and Surface Design: Saint–Malo 1999*, Vanderbilt University Press, Nashville, 2000. *BookLink (2)*

✱459 M. Fecko. *J. Math. Phys.* 36, 6709 (1995). *DOI-Link*

✱460 C. D. Ferguson, W. Klein, J. B. Rundle. *Computers Physics* 12, 34 (1998).

✱461 F. M. Fernández, R. Guardiola, J. Ros, M. Znojil. *quant-ph*/9812026 (1998). *Get Preprint*

✱462 J.-A. Ferrez, T. M. Liebling, D. Müller in K. R. Mecke, D. Stoyan (eds.). *Statistical Physics and Spatial Statis-tics*, Springer-Verlag, Berlin, 2000. *BookLink*

✶463 J. H. Field. *Am. J. Phys.* 68, 367 (2000). *DOI-Link*

✶464 J. H. Field. *arXiv:physics*/0403094 (2004). *Get Preprint*

✶465 S. B. Field, M. Klaus, M. G. Moore, F. Nori. *Nature* 388, 252 (1997). *DOI-Link*

✶466 M. Filatov, D. Cremer. *J. Chem. Phys.* 119, 11526 (2003). *DOI-Link*

✶467 J. P. Fillmore, M. Paluszny. *Seminarber. Mathematik Fernuniversität Hagen* 62, 45, (1997).

✶468 T. M. A. Fink, Y. Mao. *Physica* A 276, 109 (2000). *DOI-Link*

✶469 D. L. Finn. *Coll. Math. Mag.* 33, 283 (2002).

✶470 P. Flajolet, Y. Guivarc'h, W. Szpankowski, B. Vallée. *Preprint* (2001).
http://algo.inria.fr/flajolet/Publications/icalp01-sub.ps.gz

✶471 M. R. Flannery. *Am. J. Phys.* 73, 265 (2005). *DOI-Link*

✶472 A. P. Flitney, J. Ng, D. Abbott. *arXiv:quant-ph*/0201037 (2002). *Get Preprint*

✶473 G. Flores–Hidalgo, A. P. C. Malbouisson. *Phys. Lett.* A 337, 37 (2005). *DOI-Link*

✶474 J. L. Flowers, B. W. Petley. *Rep. Progr. Phys.* 64, 1191 (2001). *DOI-Link*

✶475 F. Fonseca, H. J. Herrmann. *arXiv:cond-mat*/0301571 (2003). *Get Preprint*

✶476 R. L. Foote. *arXiv:math.DG*/9808070 (1998). *Get Preprint*

✶477 R. L. Foote. *arXiv:math.DG*/9808070 (1998). *Get Preprint*

✶478 M. Forger, S. Sachse. *arXiv:math-ph*/9905017 (1999). *Get Preprint*

✶479 T. Fort. *Finite Differences*, Clarendon Press, Oxford, 1948. *BookLink*

✶480 J. Foster, F. B. Richards. *Am. Math. Monthly* 98, 47 (1991).

✶481 M. E. Fouladvand, Z. Sadjadi, M. R. Shaebani. *arXiv:cond-mat*/0309560 (2003). *Get Preprint*

✶482 N. D. Fowkes, J. J. Mahony. *An Introduction to Mathematical Modelling*, Wiley, Chichester, 1994.
BookLink (2)

✶483 M. P. Frank. *Comput. Sc. Eng.* n3, 16 (2002).

✶484 G. Franke, W. Suhr, F. Rieß. *Eur. J. Phys.* 11, 116 (1990). *DOI-Link*

✶485 M. Frantz. *Am. Math. Monthly* 105, 609 (1998).

✶486 L. Frappat, P. Sorba, A. Sciarrino. *arXiv:physics*/9801027 (1998). *Get Preprint*

✶487 L. Frappat, A. Sciarrino, P. Sorba. *arXiv:physics*/0003037 (2000). *Get Preprint*

✱488 L. Frappat, A. Sciarrino, P. Sorba. *arXiv:physics*/0007034 (2000). *Get Preprint*

✱489 G. N. Frederickson. *Dissections, Plane & Fancy*, Cambridge University Press, Cambridge, 1997. *BookLink (2)*

✱490 H. I. Freedman, S. D. Riemenschneider. *SIAM Rev.* 25, 561 (1983).

✱491 S. J. Freeland. *Genet. Program. Evolv. Mach.* 3, 113 (2002). *DOI-Link*

✱492 R. M. French. *Math. Intell.* 10, n4, 21 (1988).

✱493 J. Freund, T. Pöschel. *Physica* A 219, 95 (1995). *DOI-Link*

✱494 F. G. Friedlander. *Lond. Math. Soc.* 27, 551 (1973).

✱495 G. Friesecke. *Proc. R. Soc. Lond.* A 459, 47 (2003). *DOI-Link*

✱496 H. L. Frisch, C. Borzi, G. Ord, J. K. Percus, G. O. Williams. *Phys. Rev. Lett.* 63, 927 (1989). *DOI-Link*

✱497 C. Frohlich. *Am. J. Phys.* 47, 583 (1979). *DOI-Link*

✱498 N. Fuchikama, S. Ishioka, K. Kiyono. *arXiv:chao-dyn*/9811020 (1998). *Get Preprint*

✱499 P. M. Fuchs. *Math. Meth. Appl. Sci.* 14, 447 (1991).

✱500 P. M. Fuchs. *Math. Meth. Appl. Sci.* 14, 461 (1991).

✱501 P. M. Fuchs. *Math. Meth. Appl. Sci.* 18, 201 (1995).

✱502 B. Fuchssteiner, W. Wiwianka, K. Gottheil, A. Kemper, O. Kluge, K. Morisse, H. Naundorf, G. Oevel, T. Schulze. *MuPAD Multi-Processing Algebra Data Tool Benutzerhandbuch*, Birkhäuser, Basel, 1993. *BookLink*

✱503 H. Fuks. *arXiv:comp-gas*/9902001 (1999). *Get Preprint*

✱504 A. W. Fuller. *Math. Gaz.* 41, 9 (1957).

✱505 T. Fülöp, I. Tsutsui. *Phys. Lett.* A 264, 366 (2000). *DOI-Link*

✱506 W. Fulton. *Bull. Am. Math. Soc.* 37, 209 (2000). *DOI-Link*

✱507 D. Funaro. *J. Sc. Comput.* 17, 67 (2002). *DOI-Link*

✱508 Y. B. Gaididei, P. L. Christiansen, P. G. Kevrekidis, A. R. Bishop. *arXiv:nlin.PS*/0409010 (2004). *Get Preprint*

✱509 D. Gaier. *Konstruktive Methoden der konformen Abbildung*, Springer-Verlag, Berlin, 1964. *BookLink*

✱510 A. Galindo, M. A. Martín–Delgado. *arXiv:quant-ph*/0112105 (2001). *Get Preprint*

✱511 J. Gallant. *Am. J. Phys.* 70, 160 (2002). *DOI-Link*

✱512 L. Galleani, L. Cohen. *Phys. Lett.* A 302, 149 (2002). *DOI-Link*

✱513 E. Gallopoulus, E. Houstis, J. R. Rice (eds.). *Future Research Directions in Problem Solving Environments for Computational Science: Report of a Workshop on Research Directions in Integrating Numerical Analysis, Symbolic Computing, Computational Geometry, and Artificial Intelligence for Computational Science* http://www.cs.purdue.edu/research/cse/publications/tr/92/92-032.ps.gz (1991).

✱514 A. Gamliel, K. Kim, A. I. Nachman, E. Wolf. *J. Opt. Soc. Am.* A 6, 1388 (1989).

✱515 W. Gander, J. Hrebicek. *Solving Problems in Scientific Computing Using Maple and Matlab*, Springer-Verlag, Berlin, 1993. *BookLink (3)*

✱516 J. Gao, H. Cai. *Phys. Lett.* A 270, 75 (2000). *DOI-Link*

✱517 A. Garcia, M. Hubbard. *Proc. R. Soc. Lond.* A 418, 165 (1988).

✱518 M. Gardner. *Sci. Am.* 232 n4, 126 (1975).

✱519 M. Gardner. *Fractal Music, Hypercards and More*, Freeman, New York, 1992. *BookLink (2)*

✱520 T. Gardner, G. Cecchi, M. Magnasco. *Phys. Rev. Lett.* 87, 208201 (2001). *DOI-Link*

✱521 B. M. Garraway, S. Stenholm. *Phys. Rev.* A 60, 63 (1999). *DOI-Link*

✱522 P. L. Garrido, P. I. Hurtado, B. Nadrowski. *arXiv:cond-mat*/0104453 (2001). *Get Preprint*

✱523 R. Gasch, M. Lang. *ZAMM* 80, 137 (2000). *DOI-Link*

✱524 H. Gelman. *Eur. J. Phys.* 12, 230 (1991). *DOI-Link*

✱525 J. Gerhard, W. Oevel, F. Postel, S. Wehmeier. *MuPAD Tutorial*, Springer-Verlag, Berlin, 2000. *BookLink (2)*

✱526 C. Gershenson. *arXiv:nlin.AO*/0411066 (2004). *Get Preprint*

✱527 A. Gersten. *Found. Phys.* 31, 1211 (2001). *DOI-Link*

✱528 J. L. Gerver. *Geom. Dedicata* 42, 267 (1992).

✱529 H. Geyer, A. Seyfarth, R. Blickhan. *J. Theor. Biol.* 232, 315 (2005). *DOI-Link*

✱530 C. Giardiná, R. Livi, A. Politi, M. Vassalli. *Phys. Rev. Lett.* 84, 2144 (2000). *DOI-Link*

✱531 C. G. Gibson, P. E. Newstead. *Acta Appl. Math.* 7, 113 (1986). *DOI-Link*

✱532 E. N. Gilbert. *Am. Math. Monthly* 98, 201 (1991).

✱533 R. D. Gill, S. Johansen. *Ann. Stat.* 18, 1501 (1990).

✱534 N. M. Glazunov. *arXiv:math.SC*/0009057 (2000). *Get Preprint*

✱535 H. Gloggengieser. *Maple V*, Markt und Technik, Haar, 1993. *BookLink*

✱536 E. Y. Glushko. *Phys. Solid State* 38, 1132 (1996).

✱537 S. Gluzman, D. Sornette. *arXiv:cond-mat*/0106316 (2001). *Get Preprint*

✱538 J. Glynn, T. Gray. *The Beginner's Guide to Mathematica Version 4*, Cambridge University Press, Cambridge, 1999. *BookLink*

✱539 G. H. Goedecke. *Phys. Rev.* 135, B281 (1964). *DOI-Link*

✱540 J. Goldfinch. *Math. Today* 33, n2, 43 (1997).

✱541 S. Goldstein, K. A. Kelly, E. R. Speer. *J. Number Th.* 42, 1 (1992). *DOI-Link*

✱542 J. Goldstone, R. L. Jaffe. *Phys. Rev.* B 45, 14100 (1992). *DOI-Link*

✱543 G. Golse, H. Jirari, H. Kröger, K. J. M. Moriarty in G. Hunter, S. Jeffers, J.-P. Vigier (eds.). *Causality and Locality in Modern Physics*, Kluwer, Dordrecht, (1998). *BookLink*

✱544 M. A. F. Gomes, G. L. Vasconcelos, C. C. Nascimento. *J. Phys.* A 20, L1167 (1987). *DOI-Link*

✱545 J. A. González, R. Pino. *Physica* A 276, 425 (2000). *DOI-Link*

✱546 J. A. Gonzáles, L. I. Reyes, L. E. Guerrero. *arXiv:nlin.CD*/0101049 (2001). *Get Preprint*

✱547 J. A. Gonzáles, L. I. Reyes, J. J. Suárez, L. E. Guerrero, G. Gutiérrez. *Physica* A 316, 259 (2001). *DOI-Link*

✱548 E. A. González–Velasco. *Fourier Analysis and Boundary Value Problems*, Academic Press, San Diego, 1995. *BookLink*

✱549 P. R. Gordoa. *Theor. Math. Phys.* 137, 1430 (2003). *DOI-Link*

✱550 F. Gori in J. C. Dainty (ed.). *Current Trends in Optics*, Academic Press, London, 1994. *BookLink*

✱551 A. Goriely, T. McMillen. *Phys. Rev. Lett.* 88, 244301 (2002). *DOI-Link*

✱552 S. Goto, Y. Masutomi, K. Nozaki. *arXiv:patt-sol*/9905001 (1999). *Get Preprint*

✱553 S. Goto, Y. Masutomi, K. Nozaki. *Progr. Theor. Phys.* 102, 471 (1999).

✱554 D. Gottlieb, S. Orszag. *J. Comput. Appl. Math.* 43, 81 (1992). *DOI-Link*

✱555 D. Gottlieb, S. Orszag. *Comput. Meth. Appl. Mech. Eng.* 116, 27 (1994).

✱556 D. Gottlieb, S. Orszag. *Math. Comput.* 64, 1081 (1995).

✱557 D. Gottlieb, C.-W. Shu. *Num. Math.* 71, 511 (1995). *DOI-Link*

✱558 Y. Gousseau, F. Roueff. *arXiv:math.PR*/0312035 (2003). *Get Preprint*

✱559 S. Gov, S. Shtrikman. *physics*/9902002 (1999). *Get Preprint*

✱560 E. Gozzi. *arXiv:quant-ph*/0208046 (2002). *Get Preprint*

✱561 E. Gozzi, D. Mauro. *Ann. Phys.* 296, 152 (2002). *DOI-Link*

✱562 E. Gozzi, D. Mauro. *arXiv:quant-ph*/02306029 (2003). *Get Preprint*

✱563 E. Gozzi, D. Mauro, A. Silvestri. *arXiv:hep-th*/0410129 (2004). *Get Preprint*

✱564 J. Grabmeier, E. Kaltofen, V. Weisspfenning (eds.). *Computer Algebra Handbook*, Springer-Verlag, Berlin, 2002. *BookLink*

✱565 R. L. Graham, J. C. Lagarias, C. L. Mallows, A. R. Wilks, C. H. Yan. *arXiv:math.MG*/0010298 (2000). *Get Preprint*

✱566 R. L. Graham, J. C. Lagarias, C. L. Mallows, A. R. Wilks, C. H. Yan. *arXiv:math.MG*/0010302 (2000). *Get Preprint*

✱567 R. L. Graham, J. C. Lagarias, C. L. Mallows, A. R. Wilks, C. H. Yan. *arXiv:math.MG*/0010324 (2000). *Get Preprint*

✱568 R. L. Graham, J. C. Lagarias, C. L. Mallows, A. R. Wilks, C. H. Yan. *J. Number Th.* 100, 1 (2003). *DOI-Link*

✱569 P. Gralewicz, K. Kowalski. *arXiv:math-ph*/0002044 (2000). *Get Preprint*

✱570 E. Granot. *arXiv:cond-mat*/0107594 (2001). *Get Preprint*

✱571 H. L. Gray, N. F. Zhang. *Math. Comput.* 50, 513 (1988).

✱572 R. L. Greene. *Classical Mechanics with Maple*, Springer-Verlag, New York, 1995. *BookLink*

✱573 W. M. Greenlee. *Bull. Am. Math. Soc.* 82, 341 (1975).

✱574 G.-M. Greuel. *arXiv:math.AG*/0002247 (2000). *Get Preprint*

✱575 D. J. Griffith, Y. Li. *Am. J. Phys.* 64, 706 (1996). *DOI-Link*

✱576 D. Gronau in W. Förg–Rob, D. Gronau, C. Mira, N. Netzter, G. Targonsky (eds.). *Iteration Theory*, World Scientific, Singapore, 1996. *BookLink*

✱577 C. G. Grosjean. *SIAM Rev.* 38, 515 (1996).

✱578 E. K. U. Gross, R. M. Dreizler. *Density Functional Theory*, Plenum Press, New York, 1995. *BookLink (2)*

✱579 P. W. Gross, P. R. Kotiuga. *Electromagnetic Theoryand Computation: A Topological Approach*, Cambridge University Press, Cambridge, 2004. *BookLink*

✱580 M. P. Grosset, A. P. Veselov. *arXiv:math.GM*/0503175 (2005). *Get Preprint*

✱581 C. Gruber, S. Pache, A. Lesne. *arXiv:cond-mat*/0109542 (2001). *Get Preprint*

✱582 C. Gruber, S. Pache. *arXiv:cond-mat*/0204220 (2002). *Get Preprint*

✶583 C. Gruber, S. Pache, A. Lense. *J. Stat. Phys.* 117, 739 (2004). *DOI-Link*

✶584 P. M. Gruber. *Rendiconti Sem. Mat. Messina* ser II, 1, 21, (1991).

✶585 H. Grunsky. *The General Stokes' Theorem*, Pitman, Boston, 1983. *BookLink*

✶586 D. Gruntz in M. J. Wester (ed.). *Computer Algebra Systems*, Wiley, Chichester, 1999. *BookLink*

✶587 J. Guckenheimer, P. Holmes. *Nonlinear Oscillations, Dynamical Systems, and Bifurcation Vector Fields*, Springer-Verlag, New York, 1986. *BookLink*

✶588 G. G. Gunderson, L.-Z. Yang. *J. Math. Anal. Appl.* 223, 88 (1998). *DOI-Link*

✶589 P. Günther. *Huygens' Principle and Hyperbolic Equations*, Academic Press, New York, 1988. *BookLink*

✶590 Z.-H. Guo. *Science in China* A 37, 432 (1994).

✶591 L. Gurevich, V. Mostepanenko. *Phys. Lett.* A 35, 201 (1971). *DOI-Link*

✶592 E. Gutkin, P. K. Newton. *J. Phys.* A 37, 11989 (2004). *DOI-Link*

✶593 M. C. Gutzwiller. *Chaos in Classical and Quantum Mechanics*, Springer-Verlag, New York, 1990. *BookLink*

✶594 R. K. Guy in R. A. Mollin (ed.). *Number Theory Applications*, Kluwer, Dordrecht, 1989. *BookLink*

✶595 R. K. Guy. *Unsolved Problems in Number Theory*, Springer-Verlag, New York, 1994. *BookLink (3)*

✶596 W. Hackbusch. *Computing* 68, 193 (2002). *DOI-Link*

✶597 O. Haeberlé. *Opt. Commun.* 141, 237 (1997). *DOI-Link*

✶598 P. Hähner, Y. Drossinos. *Physica* A 260, 391 (1998). *DOI-Link*

✶599 B. Haible, T. Papanikolaouin in J. P. Buhler (ed.). *Algorithmic Number Theory*, Springer-Verlag, Berlin, 1998. *BookLink*

✶600 A. M. Hamel. *J. Combinat. Th.* A 94, 205 (2001). *DOI-Link*

✶601 G. Hamel. *Theoretische Mechanik. Eine einheitliche Einführung in die gesamte Mechanik*, Springer-Verlag, Berlin, 1949. *BookLink*

✶602 J. F. Hamilton, Jr., L. S. Schulman. *J. Math. Phys.* 12, 160 (1971). *DOI-Link*

✶603 S.-I Han, S. Stapf, B. Blümich. *Phys. Rev. Lett.* 87, 144501 (2001). *DOI-Link*

✶604 J. H. Hannay, G. D. Walters. *J. Phys.* A 24, L 1333 (1991). *DOI-Link*

✶605 D. B. Hanson. *Proc. R. Soc. Lond.* A 449, 315 (1995).

✶606 G. P. Harmer, D. Abbott. *Stat. Sci.* 14, 206 (1999). *DOI-Link*

✶607 G. P. Harmer, D. Abbott, P. G. Taylor, J. M. R. Parrondo. *Chaos* 11, 705 (2001). *DOI-Link*

✶608 E. M. Harrell, II. *Ann. Phys.* 105, 379 (1977). *DOI-Link*

✶609 B. Hartnell, Q. Li. *Congr. Numerantium* 145, 187 (2000).

✶610 M. B. Hastings, L. S. Levitov. *Physica* D 116, 244 (1998). *DOI-Link*

✶611 K. Hatada in J. M. Rassias (ed.). *Geometry, Analysis and Mechanics*, World Scientific, Singapore, 1995. *BookLink*

✶612 Y. Hayase. *J. Phys. Soc. Jpn.* 66, 2584 (1997). *DOI-Link*

✶613 Y. Hayase, T. Ohta. *Phys. Rev. Lett.* 81, 1726 (1998). *DOI-Link*

✶614 Y. Hayase in M. Tokuyama, H. E. Stanley. *Statistical Physics*, American Institute of Physics, Melville, 2000. *BookLink (2)*

✶615 Y. Hayase, T. Ohta. *Phys. Rev.* E 62, 5998 (2000). *DOI-Link*

✶616 Y. Hayashima, M. Nagayama, S. Nakata. *Kyoto University RIMS Technical Report* 1303 (2000). http://www.kurims.kyoto-u.ac.jp/~nagayama/PS/1303.ps

✶617 L. He, D. Vanderbilt. *arXiv:cond-mat*/0102016 (2001). *Get Preprint*

✶618 A. Heck. *Introduction to Maple*, Springer-Verlag, New York, 1993. *BookLink (2)*

✶619 M. A. Heckl, I. D. Abrahams. *J. Sound Vibr.* 229, 669 (2000). *DOI-Link*

✶620 F. W. Hehl, V. Winkelmann, H. Meyer. *REDUCE: Ein Kompaktkurs über die Anwendung von Computer-Algebra*, Springer-Verlag, Berlin, 1993. *BookLink*

✶621 D. Helbing. *Physica* A 219, 391 (1995). *DOI-Link*

✶622 D. Helbing, J. Keltsch, P. Moinár. *Nature* 388, 47 (1997). *DOI-Link*

✶623 D. Helbing, M. Treiber. *arXiv:cond-mat*/9812299 (1998). *Get Preprint*

✶624 D. Helbing in B. Kramer (ed.). *Advances in Solid State Physics* v.41, Springer-Verlag, Berlin, 2001. *BookLink*

✶625 H. G. E. Hentschel, M. N. Popescu, F. Family. *Phys. Rev.* E 65, 036141 (2002). *DOI-Link*

✶626 C. Hermann. *Acta Cryst.* 2, 139 (1949).

✶627 F. J. Herranz, M. Santander. *arXiv:math-ph*/9909005 (1999). *Get Preprint*

✶628 H. J. Herrmann, G. Sauermann. *Physica* A 283, 24 (2000). *DOI-Link*

✶629 H. J. Herrmann. *Compt. Rend. Physique* 3, 197 (2002).

✶630 P. Hersen. *arXiv:cond-mat*/0308100 (2003). *Get Preprint*

✶631 D. R. Hershbach. *Int. J. Quant. Chem.* 57, 295 (1996).

✶632 E. Hewitt, R. E. Hewitt. *Arch. Hist. Exact Sci.* 21, 129 (1979). *DOI-Link*

✶633 F. R. Hickey. *Am. J. Phys.* 47, 711 (1979). *DOI-Link*

✶634 R. Hilfer. *Applications of Fractional Calculus in Physics*, World Scientific, Singapore, 2000. *BookLink*

✶635 M. A. Hitz, E. Kaltofen, Y. N. Lakshman in S. Dooley (ed.). *ISSAC 99*, ACM Press, New York, 1999. *DOI-Link*

✶636 E. Hlawka. *Theorie der Gleichverteilung*, BI, Mannheim, 1979. *BookLink*

✶637 B. J. Hoenders, H. A. Ferwerda. *Phys. Rev. Lett.* 87, 060401-1 (2001). *DOI-Link*

✶638 T. Hogg, B. A. Hubermann, C. P. Williams. *Artif. Intell.* 81, 1 (1996). *DOI-Link*

✶639 M. H. Holmes, J. G. Ecker, W. Boyce, W. Siegmann. *Exploring Calculus with Maple*, Addison-Wesley, Reading, 1993. *BookLink*

✶640 D. C. Hong, J. A. Both. *Physica* A 289, 557 (2001). *DOI-Link*

✶641 J.-M. Hong, C.-H. Kim. *Comput. Graphics Forum* 22, 601 (2003). *DOI-Link*

✶642 M. Horbatsch. *Quantum Mechanics Using Maple*, Springer-Verlag, New York, 1995. *BookLink*

✶643 J. E. M. Hornos, Y. M. M. Hornos. *Phys. Rev. Lett.* 71, 4401 (1993). *DOI-Link*

✶644 A. Horwitz. *J. Comput. Appl. Math.* 134, 1 (2001). *DOI-Link*

✶645 A. S. Householder. *The Numerical Treatment of a Single Nonlinear Equation*, McGraw–Hill, New York, 1970. *BookLink*

✶646 P.-K. Hsiung, R. H. Thibadeau, M. Wu. *Comput. Graphics* 24, 83 (1990).

✶647 B. Hu, B. Li, H. Zhao. *arXiv:cond-mat*/0002192 (2000). *Get Preprint*

✶648 B. A. Huberman, L. A. Adamic. *arXiv:cond-mat*/9801071 (1998). *Get Preprint*

✶649 R. L. Hughes. *Math. Comput. Simul.* 53, 367 (2000). *DOI-Link*

✶650 N. Hungerbühler. *Int. J. Math. Educ.* 27, 483 (1996).

✶651 D. L. Hunter, G. A. Baker, Jr. *Phys. Rev.* B 7, 3346 (1974). *DOI-Link*

✶652 C. Hurst. *Austral. Math. Soc. Gaz.* 23, 154 (1996).

✶653 N. E. Hurt. *Mathematical Physics of Quantum Wires and Devices*, Kluwer, Dordrecht, 2000. *BookLink*

✶654 K. Iguchi. *Mod. Phys. Lett.* B 15, 981 (2001). *DOI-Link*

✶655 M. Ikawa. *Hyperbolic Differential Equations and Wave Phenomena*, American Mathematical Society, Providence, 2000. *BookLink*

✱656 M. Ikegami, Y. Nagaoka. *Progr. Theor. Phys.* S 106, 235 (2003).

✱657 A. Ilarazza-Lomelí. *arXiv:chao-dyn*/9906033 (1999). *Get Preprint*

✱658 G. K. Immink. *SIAM J. Math. Anal.* 22, 238 (1991).

✱659 L. S. Isaeva. *J. Appl. Math. Mech.* 23, 572 (1959). *DOI-Link*

✱660 J. M. Isidro. *arXiv:hep-th*/0110151 (2001). *Get Preprint*

✱661 C. J. Isham. *Lectures on Quantum Theory*, Imperial College Press, 1995. *BookLink (2)*

✱662 M. N. Islam. *Biom. J.* 37, 119 (1995).

✱663 S. Ismail-Beigi, T. A. Arias. *arXiv:cond-mat*/9909130 (1999). *Get Preprint*

✱664 M. Isobe, D. Helbing, T. Nagatani. *arXiv:cond-mat*/02306136 (2003). *Get Preprint*

✱665 G. Istrate. *arXiv:cs.CC*/0211012 (2002). *Get Preprint*

✱666 V. Ivancevic. *SIAM Rev.* 46, 455 (2004). *DOI-Link*

✱667 M. V. Ivanov. *arXiv:physics*/0206036 (2002). *Get Preprint*

✱668 O. I. Ivanova, R. K. Sabirov. *Russ. Phys. J.* 44, 454 (2001). *DOI-Link*

✱669 I. D. Ivanović. *J. Phys.* A 14, 3241 (1981). *DOI-Link*

✱670 A. Ivey, D. A. Singer. *arXiv: math.DG*/9901131 (1999). *Get Preprint*

✱671 S. Iwasaki, K. Honda. *J. Phys. Soc. Jpn.* 69, 1579 (2000). *DOI-Link*

✱672 E. Jabotinsky. *Trans. Am. Math. Soc.* 108, 457 (1963).

✱673 J. D. Jackson. *Am. J. Phys.* 68, 789 (2000). *DOI-Link*

✱674 J. D. Jackson. *Am. J. Phys.* 70, 409 (2002). *DOI-Link*

✱675 *J. ACM* 50, n1 (2003). *DOI-Link*

✱676 A. Janner. *Cryst. Eng.* 4, 119 (2001). *DOI-Link*

✱677 A. Janner. *Acta Cryst.* A 58, 334 (2002). *DOI-Link*

✱678 C. Jarzynski. *Phys. Rev.* E 56, 5018 (1997). *DOI-Link*

✱679 R. D. Jenks, R. S. Sutor. *AXIOM: The Scientific Computation System*, Springer-Verlag, New York, 1992. *BookLink*

✱680 E. T. Jensen, M. A. Shegelski. *Can. J. Phys.* 82, 791 (2004). *DOI-Link*

✶681 A. J. Jerri. *The Gibbs Phenomenon in Fourier Analysis, Splines and Wavelet Approximations*, Kluwer, Dordrecht, 1998. *BookLink*

✶682 L. Jian-cheng. *J. Math. Phys.* 29, 2254 (1988). *DOI-Link*

✶683 C. Jiang, A. W. Troesch, S. W. Shaw. *Phil. Trans. R. Soc. Lond.* A 358, 1761 (2000). *DOI-Link*

✶684 R. Jiang, B. Jia, Q.-S. Wu. *Int. J. Mod. Phys.* C 15, 619 (2004). *DOI-Link*

✶685 D. A. Jiménez-Ramírez. *Phys. Educ.* 30, 46 (1995).

✶686 M. A. Jimenéz–Montaño, C. R. de la Mora–Basáñez, T. Pöschel. *arXiv:cond-mat*/0204044 (2002). *Get Preprint*

✶687 E. Johnson. *Linear Algebra Using Maple*, Brooks/Cole, Pacific Grove, 1993. *BookLink*

✶688 R. C. Johnson. *Am. J. Phys.* 65, 296 (1997). *DOI-Link*

✶689 D. Joubert (ed.). *Density Functionals: Theory and Applications*, Springer-Verlag, Berlin, 1997. *BookLink*

✶690 S. C. Jun. *Comput. Math. Appl.* 41, 373 (2001). *DOI-Link*

✶691 H.-H. Kairies. *Aequ. Math.* 53, 207 (1997).

✶692 S. Kais, R. Bleil. *J. Chem. Phys.* 102, 7472 (1995). *DOI-Link*

✶693 H.-C. Kaiser, J. Rehberg (with an appendix by U. Krause). *WIAS Preprints* 338/199 (1997). http://vieta.wias-berlin.de/WIAS_publ_preprints_nr338.AB

✶694 H.-C. Kaiser, J. Rehberg. *ZAMP* 50, 423 (1999).

✶695 R. N. Kalia (ed.). *Recent Advances in Fractional Calculus*, Global Publishing Co., Sauk Rapids, 1993. *BookLink*

✶696 E. G. Kalnins, W. Miller, Jr. *J. Math. Phys.* 19, 1233 (1978). *DOI-Link*

✶697 E. G. Kalnins, W. Miller, Jr. *J. Math. Phys.* 19, 1247 (1978). *DOI-Link*

✶698 E. G. Kalnins, W. Miller, Jr., G. S. Pogosyan in H.-D. Doebner, S. T. Ali, M. Keyl, R. F. Werner. *Trends in Quantum Mechanics*, World Scientific, Singapore, 2000. *BookLink*

✶699 E. G. Kalnins, W. Miller, Jr., G. S. Pogosyan. *arXiv:math-ph*/0210002 (2002). *Get Preprint*

✶700 E. Kamerich. *A Guide to Maple*, Springer-Verlag, New York, 1994. *BookLink*

✶701 T. R. Kane, M. P. Scher. *Int. J. Solids Struct.* 5, 663 (1969).

✶702 L. V. Kantorovich, V. I. Krylov. *Approximate Methods of Higher Analysis*, Noordhoff, Groningen, 1964. *BookLink*

✶703 T. Kapitaniak. *Chaotic Oscillators*, World Scientific, Singapore, 1992. *BookLink (2)*

✶704 A. Kaplan, N. Friedman. M. Andersen, N. Davidson. *arXiv:nlin.CD*/0210075 (2002). *Get Preprint*

★705 M. Kapovich, J. J. Millson. *arXiv:math.AG*/9803150 (1998). *Get Preprint*

★706 R. L. Karp. *arXiv:hep-th*/0101204 (2001). *Get Preprint*

★707 S. Y. Karpov, S. N. Stolyarov. *Phys. Usp.* 36, 1 (1993).

★708 E. Kasper. *Adv. Imaging Electron Phys.* 116, 1 (2001). *BookLink*

★709 P. Kaštánek, J. Kosek, D. Šnita, I. Schreiber, M. Marek. *Physica* D 84, 79 (1995). *DOI-Link*

★710 D. Kaszlikowski, P. Gnacinski, M. Zukowski, W. Miklaszewski, A. Zeilinger. *arXiv:quant-ph*/0005028 (2000). *Get Preprint*

★711 I. Katai, B. Kovacs. *Acta Sci. Math.* 48, 221 (1985).

★712 T. Katsuyama, K. Nagata. *arXiv:chao-dyn*/9901018 (1999). *Get Preprint*

★713 A. L. Kawczynski, B. Legawiec. *Phys. Rev.* E 64, 056202 (2001). *DOI-Link*

★714 K. G. Kay. *Phys. Rev. Lett.* 83, 5190 (1999). *DOI-Link*

★715 K. G. Kay. *Phys. Rev.* A 63, 042110 (2001). *DOI-Link*

★716 K. G. Kay. *Phys. Rev.* A 65, 032101 (2002). *DOI-Link*

★717 R. J. Kay, N. F. Johnson. *arXiv:cond-mat*/0207386 (2002). *Get Preprint*

★718 S. Kehrein, C. Münkel, K. J. Wiese. *arXiv:physics*/9808038 (1998). *Get Preprint*

★719 J. B. Keller. *Am. Math. Monthly* 93, 191 (1986).

★720 J. B. Keller. *Am. J. Phys.* 71, 282 (2003). *DOI-Link*

★721 O. Keller. *Phys. Rev.* A 62, 022111 (2000). *DOI-Link*

★722 E. Kelley, M. Wu. *Phys. Rev. Lett.* 79, 1265 (1997). *DOI-Link*

★723 A. Kemnitz, M. Möller in I. Bárány, K. Böröczky (eds.). *Intuitive Geometry*, Janos Bolyai Math. Soc. Budapest, 1995. *BookLink*

★724 A. Kempf. *arXiv:gr-qc*/9907084 (1999). *Get Preprint*

★725 A. Kempf. *J. Math. Phys.* 41, 2360 (2000). *DOI-Link*

★726 A. Kempf, P. J. S. G. Ferreira. *arXiv:quant-ph*/0305148 (2003). *Get Preprint*

★727 R. D. Kent, M. Schlesinger, B. G. Wybourne. *Can. J. Phys.* 76, 445 (1998). *DOI-Link*

★728 R. Kerner. *arXiv:math-ph*/0011023 (2000). *Get Preprint*

★729 R. Kerner. *Class. Quantum Grav.* 14, A203 (1997). *DOI-Link*

★730 P. Kessler, O, M, O'Reilly. *Reg. Chaotic Dynam.* 7, 49 (2002). *DOI-Link*

★731 E. Kestemont, C. van den Broeck, M. Malek Mansour. *Europhys. Lett.* 49, 143 (2000). *DOI-Link*

★732 E. S. Key, M. M. Klosek, D. Abbott. *arXiv:math.PR*/0206151 (2002). *Get Preprint*

★733 M. Keyl, R. F. Werner. *arXiv:quant-ph*/0102027 (2001). *Get Preprint*

★734 S. A. Khan. *arXiv:physics*/0210001 (2002). *Get Preprint*

★735 A. Khare, U. Sukhatme. *arXiv:math-ph*/0112002 (2001). *Get Preprint*

★736 D. Kharitonsky, J. Gonczarowski. *Visual Comput.* 10, n10, 88 (1993).

★737 N. N. Khuri. *arXiv:hep-th*/0111067 (2001). *Get Preprint*

★738 N. N. Khuri. *Math. Phys. Anal. Geom.* 5, 1 (2002). *DOI-Link*

★739 M. Kibler in H.-D. Doebner, J.-D. Henning, T. D. Palev (eds.). *Group Theoretical Methods in Physics*, Springer-Verlag, Berlin, 1988. *BookLink*

★740 M. Kibler, P. Labastie. *arXiv:hep-th*/9409196 (1994). *Get Preprint*

★741 M. R. Kibler. *arXiv:quant-ph*/0310155 (2003). *Get Preprint*

★742 M. R. Kibler. *arXiv: quant-ph*/0503039 (2005). *Get Preprint*

★743 S. Kicovic, L. Webb, M. Crescimanno. *arXiv:physics*/0208087 (2002). *Get Preprint*

★744 G. Kielau, P. Maißer. *Multibody System Dyn.* 9, 213 (2003). *DOI-Link*

★745 K. Kim, E. Wolf. *Opt. Commun.* 59, 1 (1986). *DOI-Link*

★746 S.-H. Kim. *Acta Appl. Math.* 73, 275 (2002). *DOI-Link*

★747 J. C. Kimball, H. L. Frisch. *Phys. Rev. Lett.* 93, 093001 (2004). *DOI-Link*

★748 C. Kimberling. *Congr. Numer.* 129 (1998).

★749 H. C. King. *arXiv:math.AG*/9807023 (1998). *Get Preprint*

★750 H. C. King. *arXiv:math.AG*/9810130 (1998). *Get Preprint*

★751 H. C. King. *arXiv:math.AG*/9811138 (1998). *Get Preprint*

★752 A. Kiejna, K. F. Wojciechowski. *Metal Surface Electron Physics*, Elsevier, Kidlington, 1996. *BookLink*

★753 S. A. King, R. E. Parent. *J. Visual. Comput. Anim.* 12, 107 (2001). *DOI-Link*

★754 L. Kirby, J. Paris. *Bull. Lond. Math. Soc.* 14, 285 (1982).

★755 A. Kirchner, K. Nishinari, A. Schadschneider. *arXiv:cond-mat*/0209383 (2002). *Get Preprint*

✶756 D. Kirkpatrick, B. Mishra. *Discr. Comput. Geom.* 7, 295 (1992).

✶757 S. Kirkpatrick, B. Selman in N. P. Ong, R. N. Bhatt (ed.). *More Is Different*, Princeton University Press, Princeton, 2001. *BookLink*

✶758 A. Kirsch. *Math. Semesterber.* 37, 216 (1990).

✶759 V. V. Kisil. *J. Phys.* A 37, 183 (2003). *DOI-Link*

✶760 K. Kiyono, N. Fuchikami. *arXiv:chao-dyn*/9904012 (1999). *Get Preprint*

✶761 K. Kiyono, T. Katsuyama, T. Masunaga, N. Fuchikami. *arXiv:nlin.CD*/0210003 (2002). *Get Preprint*

✶762 M. S. Klamkin, D. J. Newman. *Am. Math. Monthly* 78, 631 (1971).

✶763 A. Klappenecker, M. Rötteler. *arXiv:quant-ph*/0309120 (2003). *Get Preprint*

✶764 A. Klappenecker, M. Rötteler. *arXiv:quant-ph*/0502138 (2005). *Get Preprint*

✶765 J. R. Klauder. *Acta Physica Austriaca* Suppl.XI, 341 (1973).

✶766 J. R. Klauder. *Science* 199, 735 (1978).

✶767 F. Kleefeld. *arXiv:hep-th*/0408028 (2004). *Get Preprint*

✶768 F. Klein. *Elementarmathematik vom höheren Standpunkte*, v.1, Springer-Verlag, Berlin, 1924. *BookLink*

✶769 V. B. Kokshenev. *arXiv:physics*/0404089 (2004). *Get Preprint*

✶770 D. Knuth. *Electr. J. Combinat.* 3, R5 (1996).
http://www.combinatorics.org/Volume_3/volume3_2.html#R5

✶771 A. Knutson. *arXiv:math.LA*/9911088 (1999). *Get Preprint*

✶772 A. Knutson, T. Tao. *arXiv:math.RT*/0009048 (2000). *Get Preprint*

✶773 A. Knutson, T. Tao. *Notices Am. Math. Soc.* 48, 175 (2001).

✶774 A. Knutson, T. Tao, C. Woodward. *arXiv:math.CO*/0107011 (2001). *Get Preprint*

✶775 R. Kobayashi, T. Ohta, Y. Hayase. *Phys. Rev.* E 50, R3291 (1994). *DOI-Link*

✶776 N. Köckler. *Numerical Methods and Scientific Computing*, Clarendon Press, Oxford, 1994. *BookLink*

✶777 M. Kofler. *Maple V, Release 2 Einführung und Leitfaden für den Praktiker*, Addison-Wesley, Bonn, 1993. *BookLink*

✶778 W. Kohn. *Phys. Rev.* 115, 809 (1959). *DOI-Link*

✶779 Y. Komatu. *Proc. Imp. Acad. Tokyo* 20, 536 (1944).

✶780 Y. Komatu. *Jap. J. Math.* 19, 203 (1945).

★781 H. Konno, P. S. Lomdahl. *J. Phys. Soc. Jpn.* 69, 1629 (2000).

★782 H. Koppe, A. Huber in H. P. Dürr (ed.). *Quanten und Felder*, Vieweg, Braunschweig, 1971. *BookLink*

★783 W. S. Koon, J. E. Marsden. *Rep. Math. Phys.* 40, 21 (1997). *DOI-Link*

★784 B. O. Koopman. *Proc. Natl. Acad. Sci.* 17, 315 (1931).

★785 D. J. Korteweg. *Nieuw Archief Wiskunde* 4, 130 (1899).

★786 R. Kotowski. *Z. Phys.* B 33, 321 (1979).

★787 L. P. Kouwenhoven, D. G. Austing, S. Tarucha. *Rep. Progr. Phys.* 64, 701 (2001). *DOI-Link*

★788 K. Kowalski, K. Podlaski, J. Rembielinksi. *arXiv:quant-ph*/0206176 (2002). *Get Preprint*

★789 V. V. Kozlov. *Reg. Chaotic Dynam.* 7, 161 (2002). *DOI-Link*

★790 K. W. Kratky. *J. Phys.* A 11, 1017 (1978). *DOI-Link*

★791 U. Kraus. *Am. J. Phys.* 68, 56 (2000). *DOI-Link*

★792 L. Krauss, G. D. Starkman. *arXiv:astro-ph*/0404510 (2004). *Get Preprint*

★793 P. Krehl, S. Engemann, D. Schwenkel. *Shock Waves* 8, 1 (1998). *DOI-Link*

★794 H. Kröger, S. Lantagne, K. J. M. Moriarty, B. Plache. *Phys. Lett.* A 199, 299 (1994). *DOI-Link*

★795 H. Kröger. *Phys. Rep.* 323, 81 (2000). *DOI-Link*

★796 K. Kroy, G. Sauermann, H. J. Herrmann. *arXiv:cond-mat*/0101380 (2001). *Get Preprint*

★797 K. Kroy, G. Sauermann, H. J. Herrmann. *arXiv:cond-mat*/0203040 (2002). *Get Preprint*

★798 G. P. Kubalski, M. Napiórkowski. *arXiv:cond-mat*/0008386 (2000). *Get Preprint*

★799 M. Kucma. *Functional Equations in a Single Variable*, PWN, Warszawa, 1968. *BookLink*

★800 R. Kühne. *Phys. Blätter* 47, 201 (1991).

★801 A. S. Kuleshov. *J. Appl. Math. Mech.* 65, 171 (2001). *DOI-Link*

★802 T. Kunihiro. *Progr. Theor. Phys.* 94, 503 (1995).

★803 T. Kunihiro. *Jpn. J. Indust. Appl. Math.* 14, 51 (1997).

★804 T. Kunihiro. *arXiv:hep-th*/9801196 (1998). *Get Preprint*

★805 W. Kutzelnigg. *Chem. Phys.* 225, 203 (1997). *DOI-Link*

★806 M. Kuwamura. *Jpn. J. Industr. Appl. Math.* 18, 739 (2001).

✱807 F. Kuypers, G. P. Meyer, J. Freihart, C. Friedl, M. Gerisch, H. J. Kraus, K. Seidel. *ZAMM* 74, 503 (1994).

✱808 A. V. Kuzhel, S. A. Kuzhel. *Regular Extensions of Hermitian Operators*, VSP, Utrecht, 1998. *BookLink*

✱809 G. F. Kventsel, J. Katriel. *Phys. Rev.* A 24, 2299 (1981). *DOI-Link*

✱810 G. Labelle. *Eur. J. Combinat.* 1, 113 (1980).

✱811 K. Lake. *arXiv:gr-qc*/9803072 (1998). *Get Preprint*

✱812 J. Lam. *J. Math. Phys.* 8, 1053 (1967). *DOI-Link*

✱813 L. Lambe. *Notices Am. Math. Soc.* 41, 14 (1994).

✱814 D. Lambert, M. Kibler. *J. Phys.* A 21, 307 (1988). *DOI-Link*

✱815 S. Landau. *Notices Am. Math. Soc.* 46, 189 (1999).

✱816 D. Langbein. *J. Phys.* A 10, 1031 (1986). *DOI-Link*

✱817 D. Langbein. *Capillary Surfaces*, Springer-Verlag, Berlin, 2002. *BookLink*

✱818 J. Langer, D. A. Singer. *SIAM Rev.* 38, 605 (1996).

✱819 J. S. Langer in V. L. Fitch, D. R. Marlow, M. A. E. Dementi (eds.). *Critical Problems in Physics*, Princeton University Press, Princeton, 1996. *BookLink (2)*

✱820 V. Latora, A. Rapisarda, C. Tsallis, M. Baranger. *arXiv:cond-mat*/9907412 (1999). *Get Preprint*

✱821 H. T. Lau. *A Numerical Library in C for Scientists and Engineers*, CRC Press, Boca Raton, 1995. *BookLink*

✱822 P. Leboeuf, A. G. Monastra, O. Bohigas. *Reg. Chaotic Dynam.* 6, 205 (2001). *DOI-Link*

✱823 J. Lebowitz, J. Piasecki, Y. G. Sinai. *Dokl. Math.* 62, 398 (2000).

✱824 T. Lee (ed.). *Mathematical Computations with Maple V*, Birkhäuser, Basel, 1993. *BookLink*

✱825 H. S. Leff. *Am. J. Phys.* 67, 1114 (1999). *DOI-Link*

✱826 R. I. Leine, C. Glocker, D. H. van Kampen. *Proc. ASME 2001 Design Engineering Technical Conference*, Pittsburg, 2001.

✱827 R. Leis. *Math. Meth. Appl. Sci.* 24, 339 (2001). *DOI-Link*

✱828 J. Lekner. *Math. Mag.* 55, 26 (1982).

✱829 J. Lekner. *J. Opt.* A 5, L15 (2003). *DOI-Link*

✱830 J. Lekner. *J. Opt.* A 6, 146 (2004). *DOI-Link*

✱831 J. Lekner. *J. Opt.* A 6, 711 (2004). *DOI-Link*

✱832 J. León. *arXiv:quant-ph*/0309049 (2003). *Get Preprint*

✱833 U. Leonhardt. P. Piwnicki. *arXiv:physics*/9906038 (1999). *Get Preprint*

✱834 U. Leonhardt, P. Piwnicki. *Phys. Rev. Lett.* 84, 822 (2000). *DOI-Link*

✱835 U. Leonhardt. *arXiv:gr-qc*/0108085 (2001). *Get Preprint*

✱836 J. Lepak, M. Crescimanno. *arXiv:physics*/0201053 (2002). *Get Preprint*

✱837 S. Lepri, R. Livi, A. Politi. *Phys. Rep.* 377, 1 (2003). *DOI-Link*

✱838 M. Lesser. *Int. J. Bifurc. Chaos* 4, 521 (1994). *DOI-Link*

✱839 H. Leutwyler. *Eur. J. Phys.* 15, 59 (1994). *DOI-Link*

✱840 A. Levermann, I. Procaccia. *arXiv:cond-mat*/0305552 (2003). *Get Preprint*

✱841 M. Levi, W. Weckesser. *Ergod. Th. Dynam. Sys.* 22, 1497 (2002). *DOI-Link*

✱842 A. D. Lewis, R. M. Murray. *Int. J. Nonl. Mech.* 30, 793 (1995). *DOI-Link*

✱843 B. Li, H. Zhao, B. Hu. *Phys. Rev. Lett.* 86, 63 (2001). *DOI-Link*

✱844 J. Li. *J. Opt. Soc. Am.* A 13, 1870 (1996).

✱845 T. Liang, T. A. Witten. *arXiv:cond-mat*/0407466 (2004). *Get Preprint*

✱846 K. G. Libbrecht. *Rep. Progr. Phys.* 68, 855 (2005). *DOI-Link*

✱847 S. Lien, T. Kajiya. *IEEE Comput. Graph. Appl.* Oct. 35, (1984).

✱848 G. Liger–Belair. *Ann. Phys. Fr.* 27, n4, 1 (2002). *DOI-Link*

✱849 A. R. Lima, G. Sauermann, H. J. Herrmann, K. Kroy. *Physica* A 310, 487 (2002). *DOI-Link*

✱850 F. Lindemann. *Math. Annalen* 19, 517 (1882).

✱851 B. Linet. *arXiv:gr-qc*/0011018 (2000). *Get Preprint*

✱852 A. G. Lisi. *arXiv:physics*/9907041 (1999). *Get Preprint*

✱853 R. G. Littlejohn, M. Reinsch. *Rev. Mod. Phys.* 69, 213 (1997). *DOI-Link*

✱854 E. T. Littlewood, J. E. Littlewood. *Proc. Lond. Math. Soc.* 43, 324 (1937).

✱855 F. L. Litvin. *Gear Geometry and Applied Theory*, Prentice Hall, Englewood Cliffs, 1994. *BookLink (2)*

✱856 S. S. Lo, D. A. Morales. *Int. J. Quant. Chem.* 88, 263 (2002). *DOI-Link*

✱857 R. B. Lockhardt, M. J. Steiner. *Phys. Rev.* A 65, 022107 (2002). *DOI-Link*

✱858 R. J. Lopez (ed.). *Maple V: Mathematics and Its Applications*, Birkhäuser, Boston, 1994. *BookLink*

✱859 R. J. Lopez. *Maple via Calculus*, Birkhäuser, Basel, 1994. *BookLink*

✱860 S. Lloyd. *arXiv:quant-ph*/9908043 (1999). *Get Preprint*

✱861 S. Lloyd, V. Giovannetti, L. Maccone. *Phys. Rev. Lett.* 93, 100501 (2004). *DOI-Link*

✱862 J. T. Londergan, J. P. Carini, D. P. Murdock. *Binding and Scattering in Two-Dimensional Systems*, Springer-Verlag, Berlin, 1999. *BookLink*

✱863 S. Lovejoy, M. Lilley, N. Desaulniers–Soucy, D. Schertzer. *Phys. Rev.* E 68, 025301 (2003). *DOI-Link*

✱864 M. V. Lowson. *Proc. R. Soc. Lond.* A 286, 559 (1965).

✱865 D. W. Lozier. *J. Comput. Appl. Math.* 66, 345 (1996). *DOI-Link*

✱866 R. C. Lua, A. Y. Grosberg. *arXiv:cond-mat*/0502434 (2005). *Get Preprint*

✱867 I. Lubashevsky, S. Kalenkov, R. Mahnke. *arXiv:cond-mat*/0111121 (2001). *Get Preprint*

✱868 F. Luca. *Arch. Math.* 74, 269 (2000). *DOI-Link*

✱869 F. Luccio, L. Pagli. *SIGACT News* 31, n4, 130 (2000). *DOI-Link*

✱870 R. Lück. *Mat. Sc. Eng.* A 294, 263 (2000). *DOI-Link*

✱871 C. Lunkes, Č. Brukner, V. Vedral. *arXiv:quant-ph*/0410166 (2004). *Get Preprint*

✱872 C. Lunkes, Č. Brukner, V. Vedral. *arXiv:quant-ph*/0502122 (2005). *Get Preprint*

✱873 B. Luque, R. V. Solé. *Physica* A 284, 33 (2000). *DOI-Link*

✱874 J.-G. Luque, J.-Y. Thibon. *Adv. Appl. Math.* 29, 620 (2002). *DOI-Link*

✱875 J. Lützen in J. R. Stefánson (ed.). *Proc. 19th Nordic Congress of Mathematics.*, University of Iceland, Rekjavík, 1985.

✱876 D. Lynden-Bell. *arXiv:cond-mat*/9812172 (1998). *Get Preprint*

✱877 N. MacDonald. *REDUCE for Physicists*, World Scientific, Singapore, 1994. *BookLink*

✱878 J. Maddox. *Nature* 364, 385 (1993). *DOI-Link*

✱879 E. L. Madsen. *Am. J. Phys.* 45, 182 (1977). *DOI-Link*

✱880 C. Maes. *J. Stat. Phys.* 95, 367 (1999). *DOI-Link*

✱881 L. I. Magarill, M. V. Éntin. *JETP* 96, 766 (2003). *DOI-Link*

✱882 L. Mahadevan, H. Aref, S. W. Jones. *Phys. Rev. Lett.* 75, 1420 (1995). *DOI-Link*

✱883 R. Mahnke, J. Kaupuzs, I. Lubashevsky. *Phys. Rep.* 408, 1 (2005). *DOI-Link*

✱884 J. Main. *arXiv:chao-dyn*/9902008 (1999). *Get Preprint*

✱885 M. Maioli. *J. Math. Phys.* 22, 1952 (1981). *DOI-Link*

✱886 M. Majewski. *MuPAD Pro Computing Essentials*, Springer-Verlag, Berlin, 2002. *BookLink*

✱887 P. M. Mäkilä. *Physica* D 198, 309 (2004). *DOI-Link*

✱888 L. Makkonen. *Phil. Trans. R. Soc. Lond.* A 358, 2913 (2000). *DOI-Link*

✱889 K. Malarz, S. Kaczanowska, K. Kulakowski. *arXiv:cond-mat*/0204509 (2002). *Get Preprint*

✱890 E. B. Manoukian, S. Sukkhasena. *Eur. J. Phys.* 23, 103 (2002). *DOI-Link*

✱891 S. C. Manrubia, D. H. Zanette. *arXiv:cond-mat*/0201559 (2002). *Get Preprint*

✱892 M. M. Mansour, C. van den Broeck, E. Kestemont. *Europhys. Lett.* 69, 510 (2005). *DOI-Link*

✱893 N. H. March in S. Lundquist, N. H. March (eds.). *Theory of the Inhomogeneous Electron Gas*, Plenum, New York, 1983. *BookLink*

✱894 N. H. March, S. Kais. *Int. J. Quant. Chem.* 65, 411 (1997). *DOI-Link*

✱895 S. Marconi, B. Chopard in S. Bandini, B. Chopard, M. Tomassini (eds.). *Cellular Automata*, Springer-Verlag, Berlin, 2002. *BookLink*

✱896 L. Mardoyan. *quant-ph*/0302162 (2003). *Get Preprint*

✱897 L. Mardoyan. *arXiv:quant-ph*/0308097 (2003). *Get Preprint*

✱898 P. Marecki, N. Spzak. *arXiv:quant-ph*/0407186 (2004). *Get Preprint*

✱899 E. A. Marengo, R. W. Ziolkowski. *Phys. Rev. Lett.* 83, 3345 (1999). *DOI-Link*

✱900 M. Marengo, R. Scardovelli, C. Josserand, S. Zaleski in R. Salvi (ed.). *The Navier–Stokes Equations: Theory and Numerical Methods*, Marcel Dekker, New York, 2002. *BookLink*

✱901 X. Markenscoff, L. Ni, C. H. Papadimitriou. *Int. J. Robot. Res.* 9 (1990).

✱902 P. A. Markowich, N. J. Mauser, F. Poupaud. *J. Math. Phys.* 35, 1066 (1994). *DOI-Link*

✱903 A. Marigo, A. Bicchi in G. Ferreyra, R. Gardner, H. Hermes, H. Suessmann (eds.). *Differential Geometry and Control*, American Mathematical Society, Providence, 1999. *BookLink*

✱904 J. E. Marsden in Y. Eliashberg, L. Traynor (eds.). *Symplectic Geometry and Topology*, American Mathematical Society, Providence, 1999. *BookLink*

✱905 G. Martin. *arXiv:math.NT*/9807108 (1998). *Get Preprint*

✱906 G. Martin. *arXiv:math.NT*/0206166 (2002). *Get Preprint*

✱907 H. Martini in O. Giering, J. Hoschek (eds.). *Geometrie und ihre Anwendungen,* Carl Hanser, München, 1994. *BookLink*

✱908 N. Marwan, J. Kurths in C. V. Benton (ed.). *Mathematical Physics Research at the Cutting Edge*, Nova Science, New York, 2004. *BookLink*

✱909 K. Matan, R. Williams, T. A. Witten, S. R. Nagel. *arXiv:cond-mat*/0111095 (2001). *Get Preprint*

✱910 *MathPAD* 3, n1 (1993).

✱911 G. E. A. Matsas. *Phys. Rev.* D 68, 027701 (2003). *DOI-Link*

✱912 T. Matsumoto, Y. Aizawa. *Progr. Theor. Phys.* 102, 909 (1999).

✱913 K. Matsumura, Y. Taguchi. *arXiv:nlin.PS*/0305035 (2003). *Get Preprint*

✱914 S. Matsutani. *arXiv:math.DG*/0008153 (2000). *Get Preprint*

✱915 R. A. J. Matthews. *Eur. J. Phys.* 16, 172 (1995). *DOI-Link*

✱916 C. Mattiussi in P. W. Hawkes (ed.). *Adv. Imaging Electron Phys.* 113, 1 (2000). *BookLink*

✱917 C. Mattiussi in P. W. Hawkes (ed.). *Adv. Imaging Electron Phys.* 121, 143 (2000). *BookLink*

✱918 S. M. Maurer, B. A. Huberman. *arXiv:nlin.CD*/0003041 (2000). *Get Preprint*

✱919 D. Mauro. *Int. J. Mod. Phys.* A 17, 1301 (2002). *DOI-Link*

✱920 D. Mauro. *arXiv:quant-ph*/0208190 (2002). *Get Preprint*

✱921 D. Mauro. *arXiv:quant-ph*/0301172 (2003). *Get Preprint*

✱922 G. Maze, L. Minder. *arXiv:math.GM*/0409014 (2004). *Get Preprint*

✱923 O. Mazonka, C. Jarzynski. *arXiv:cond-mat*/9912121 (2005). *Get Preprint*

✱924 A. Mazzolo, B. Roesslinger. *J. Math. Phys.* 44, 6195 (2003). *DOI-Link*

✱925 H. McGee, J. McInerney, A. Harrus. *Phys. Today.* 52, n11, 30 (1999).

✱926 B. D. McKay, N. D. Megill, M. Pavičić. *Int. J. Theor. Phys.* 39, 2381 (2000). *DOI-Link*

✱927 P. J. McKenna. *Am. Math. Monthly* 106, 1 (1999).

✱928 R. I. McLachlan, B. Ryland. *arXiv:math-ph*/0210030 (2002). *Get Preprint*

✱929 T. A. McMahon. *J. Appl. Physiol.* 39, 619 (1975).

✱930 T. McMillen, A. Goriely. *Physica* D 184, 192 (2003). *DOI-Link*

✱931 L. R. Mead, F. W. Bentrem. *Am. J. Phys.* 66, 202 (1998). *DOI-Link*

✱932 A. Mehta, G. C. Barker. *Rep. Progr. Phys.* 57, 383 (1994). *DOI-Link*

✱933 S. Mertens. *Phys. Rev. Lett.* 84, 1347 (2000). *DOI-Link*

✱934 S. Mertens. *arXiv:cond-mat*/0009230 (2000). *Get Preprint*

✱935 P. Meurs, C. an den Broeck, A. Garcia. *arXiv:cond-mat*/0407180 (2004). *Get Preprint*

✱936 D. A. Meyer, H. Blumer. *arXiv:quant-ph*/0110028 (2001). *Get Preprint*

✱937 D. A. Meyer, H. Blumer. *arXiv:quant-ph*/0110028 (2001). *Get Preprint*

✱938 R. E. Meyer. *SIAM Rev.* 31, 435 (1989).

✱939 M. Mézard, G. Parisi, R. Zecchina. *Science* 297, 812 (2002). *DOI-Link*

✱940 G. A. Mezincescu. *arXiv:quant-ph*/0002056 (2000). *Get Preprint*

✱941 T.-D. Miao, Q.-S. Mu, S.-Z. Wu. *Phys. Lett.* A 288, 16 (2001). *DOI-Link*

✱942 R. Michaels. *Eureka* 53, 16 (1994).

✱943 B. Michalowski. *SIAM Rev.* 37, 241 (1995).

✱944 W. R. E. Miguel, J. G. Pereira. *arXiv:gr-qc*/0006098 (2000). *Get Preprint*

✱945 Z. Mihailović, M. Rajković. *arXiv:cond-mat*/0307212(2003). *Get Preprint*

✱946 M. Milgrom. *arXiv:cond-mat*/9803060 (1998). *Get Preprint*

✱947 G. Millington. *Radio Sci.* 4, 95 (1969).

✱948 D. P. Minor. *Coll. Math. J.* 34, 15 (2003).

✱949 B. Mishra, J. T. Schwartz, M. Sharir. *Algorithmica* 2, 541 (1987).

✱950 B. Mishra in C.A. Gorini (ed.). *Geometry at Work: Papers in Applied Geometry*, Mathematical Association of America, New York, 2000. *BookLink*

✱951 K. A. Mitchell. *arXiv:quant-ph*/0001059 (2000). *Get Preprint*

✱952 I. M. Mladenov. *Comt. Rend. Bulg. Sc.* 54, 39 (2001).

✱953 I. M. Mladenov, J. Oprea. *Am. Math. Monthly* 110, 761 (2003).

✱954 H. K. Moffatt. *Nature* 404, 834 (2000). *DOI-Link*

✱955 K. H. Moffatt in H. Aref, J. W. Philips (eds.). *Mechanics for a New Millennium*, Kluwer, Dordrecht, 2001. *BookLink*

✱956 H. K. Moffatt, Y. Shimomura. *Nature* 416, 385 (2002). *DOI-Link*

✱957 A. Mogilner, L. Edelstein–Keshet. *J. Math. Biol.* 38, 534 (1999). *DOI-Link*

✱958 H. Momiji, S. R. Bishop, R. Carretero-González, A.Warren. *mp_arc* 00-160 (2000).
http://rene.ma.utexas.edu/mp_arc-bin/mpa?yn=00-160

✱959 R. Monasson, R. Zecchina. *Phys. Rev.* E 56, 1357 (1997). *DOI-Link*

✱960 G. Monsivais. *Am. J. Phys.* 72, 1178 (2004). *DOI-Link*

✱961 R. Montgomery. *Commun. Math. Phys.* 128, 565 (1990).

✱962 R. Montgomery in J. Enos (ed.). *Dynamics and Control of Dynamical Systems*, American Mathematical Society, Providence, 1993. *BookLink*

✱963 G. P. Morriss, C. Gruber. *J. Stat. Phys.* 109, 549 (2002). *DOI-Link*

✱964 J. D. Morrison, R. E. Moss. *Molec. Phys.* 41, 491 (1980).

✱965 H. E. Moses. *SIAM J. Appl. Math.* 21, 114 (1971).

✱966 H. E. Moses. *Ann. Henri Poincaré* 3, 773 (2002). *DOI-Link*

✱967 H. E. Moses. *Ann. Henri Poincaré* 3, 793 (2002). *DOI-Link*

✱968 H. E. Moses. *J. Math. Phys.* 45, 1887 (2004). *DOI-Link*

✱969 A. Mostafazadeh. *arXiv:quant-ph*/0407213 (2004). *Get Preprint*

✱970 A. Mostafazadeh, A. Batal. *arXiv:quant-ph*/0408132 (2004). *Get Preprint*

✱971 G. Mougin, J. Magnaudet. *Phys. Rev. Lett.* 88, 014502 (2002). *DOI-Link*

✱972 T. Mullin, A. Champneys, W. B. Fraser, J. Galan, D. Acheson. *Proc. R. Soc. Lond.* A 459, 539 (2003).
DOI-Link

✱973 I. Müller and P. Strehlow. *Rubber and Rubber Balloons: Paradigms of Thermodynamics*, Springer-Verlag, Berlin, 2004. *BookLink*

✱974 M. Müller, D. Schleicher. *arXiv:math.GM*/0502109 (2005). *Get Preprint*

✱975 C. B. Muratov. *arXiv:adap-org*/9706005 (1997). *Get Preprint*

✱976 C. B. Muratov. *arXiv:patt-sol*/9901003 (1999). *Get Preprint*

✱977 T. Murayama. *J. Phys.* A 35, L95 (2002). *DOI-Link*

✱978 M. Musette, C. Verhoeven. *Physica* D 144, 211 (2000). *DOI-Link*

✱979 G. Mussardo. *arXiv:cond-mat* 9712010 (1997). *Get Preprint*

✱980 A. Naftalevitch. *Mich. Math. J.* 22, 205 (1975).

✱981 A. Nagai. *arXiv:nlin.SI*/0206018 (2002). *Get Preprint*

✱982 K. Nagai, Y. Sumino, H. Kitahata, K. Yoshikawa. *arXiv:nlin.AO*/0502045 (2005). *Get Preprint*

✱983 T. Nagasawa, M. Sakamoto, K. Takenaga. *arXiv:hep-th*/0212192 (2002). *Get Preprint*

✱984 K. Nagel, H. J. Herrmann. *Physica* A, 199, 254 (1993). *DOI-Link*

✱985 K. Nagel. *Int. J. Mod. Phys.* C 5, 567 (1994). *DOI-Link*

✱986 D. Nagy. *Geophysics* 31, 362 (1966). *DOI-Link*

✱987 T. Nagylaki. *J. Math. Biol.* 44, 253 (2002). *DOI-Link*

✱988 S. Nakata, Y. Iguchi, S. Ose, M. Kuboyama, T. Ishii, K. Yoshikawa. *Langmuir* 13, 4454 (1997).

✱989 S. Nakata, Y. Hayashima. *J. Chem. Soc. Faraday Trans.* 94, 3655 (1998).

✱990 R. Narevich, R. E. Prange, O. Zaitsev. *arXiv:nlin.CD*/0003009 (2000). *Get Preprint*

✱991 H. L. Neal. *Am. J. Phys.* 66, 512 (1998). *DOI-Link*

✱992 T. Négadi. *Int. J. Quant. Chem.* 91, 651 (2003). *DOI-Link*

✱993 J. I. Neimark, N. A. Fufaev. *Dynamics of Anholonomic Systems*, American Mathematical Society, Providence, 1972.

✱994 A. I. Neishadt, Y. G. Sinai. *J. Stat. Phys.* 116, 815 (2004). *DOI-Link*

✱995 R. A. Nelson. *J. Math. Phys.* 35, 6224 (1994). *DOI-Link*

✱996 A. Nersessian. *arXiv:math-ph*/0010049 (2000). *Get Preprint*

✱997 J. Neubüser, H. Wondratschek, R. Bülow.*Acta Cryst.* A 27, 517 (1971).

✱998 S. Neukirch, G. H. M. van der Heijden, J. M. T. Thompson. *J. Mech. Phys. Solids* 50, 1175 (2002). *DOI-Link*

✱999 Y. J. Ng. *arXiv:gr-qc*/0006105 (2000). *Get Preprint*

✱1000 Y. J. Ng. *arXiv:hep-th*/0010234 (2000). *Get Preprint*

✱1001 R. A. Nicolaides, N. J. Walkington. *Maple—A Comprehensive Introduction*, Cambridge University Press, Cambridge, 1996. *BookLink*

✱1002 A. J. Niemi. *Proc. Steklov Inst. Math.* 226, 217 (1999).

✱1003 A. G. Nikitin. *J. Phys.* A 31, 3297 (1998). *DOI-Link*

✱1004 H. Nikolić. *Am. J. Phys.* 67, 1007 (1999). *DOI-Link*

✱1005 N. K. Nikolova, Y. S. Rickard. *Phys. Rev.* E 71, 016617 (2005). *DOI-Link*

✱1006 H. Nishimori, N. Ouchi. *Phys. Rev. Lett.* 71, 197 (1993). *DOI-Link*

✱1007 H. Nishimori, M. Yamasaki, K. H. Andersen. *Int. J. Mod. Phys.* B 12, 257 (1998). *DOI-Link*

✱1008 H. Nishimori, H. Tanaka. *arXiv:nlin.PS*/0007029 (2000). *Get Preprint*

✱1009 M. A. Nowak, N. L. Komarova, P. Niyogi. *Science* 291, 114 (2001). *DOI-Link*

✱1010 A. Nowojewski, J. Kallas, A. Dragan. *Am. Math. Monthly* 111, 817 (2004).

✱1011 K. Nozaki, Y. Oono. *Phys. Rev.* E 63, 046101 (2001). *DOI-Link*

✱1012 H. N. Núñez-Yépez, J. Delgado, A. L. Salas-Brito in A. Anzaldo-Meneses, B. Bonnard, J. P. Gauthier, F. Monroy-Pérez (eds.). *Contemporary Trends in Nonlinear Geometric Control Theory*, World Scientific, Singapore, 2002. *BookLink*

✱1013 F. Oberhettinger, W. Magnus. *Anwendungen der elliptischen Funktionen in Physik and Technik*, Springer-Verlag, Berlin, 1949. *BookLink*

✱1014 S. O’Brien, J. L. Synge. *Proc. Roy. Irish Acad.* A 56, 23 (1954).

✱1015 S. G. B. M. O’Brien. *Quart. J. Appl. Math.* 52, 43 (1994).

✱1016 A. M. Odlyzko, B. Poonen. *L’Enseignement Mathematique* 39, 317 (1993).

✱1017 A. Odlyzko. *Proc. Organic Math. Workshop 1995* (1996).
http://www.cecm.sfu.ca/organics/papers/odlyzko/paper/html/paper.html

✱1018 M. J. O’Donnell. *arXiv:cs.OH*/9911010 (1999). *Get Preprint*

✱1019 N. Ogawa. *arXiv:cond-mat*/9907381 (1999). *Get Preprint*

✱1020 N. Ogawa, Y. Furukawa. *arXiv:cond-mat*/0110392 (2001). *Get Preprint*

✱1021 N. Ogawa. *arXiv:quant-ph*/0211181 (2002). *Get Preprint*

✱1022 S. Oh, J. Kim. *Phys. Rev.* A 69, 054305 (2004). *DOI-Link*

✱1023 J. O'Hara. *Sugaku Exp.* 13, 73 (2000).

✱1024 S. Ohno. *Proc. Natl. Acad. Sci. USA* 93, 15276 (1996).

✱1025 T. Ohta, Y. Hayase, R. Kobayashi. *Phys. Rev.* E 54, 6074 (1996). *DOI-Link*

✱1026 R. O’Keefe. *Am. J. Phys.* 62, 299 (1994). *DOI-Link*

✱1027 K. Okunishi, Y. Hieida, Y. Akutsu. *Phys. Rev.* E 59, R6227 (1999). *DOI-Link*

✱1028 K. B. Oldham, J. Spanier. *The Fractional Calculus*, Academic Press, New York, 1974. *BookLink*

✱1029 O. Olendski. *Phys. Rev.* B 66, 035331 (2002). *DOI-Link*

★1030 W. M. Oliva. *Geometric Mechanics*, Springer-Verlag, Berlin, 2002. *BookLink*

★1031 P. M. Oliveira. *J. Franklin Inst.* 337, 303 (2000). *DOI-Link*

★1032 E. A. Olszewski. *arXiv:physics*/0503210 (2005). *Get Preprint*

★1033 R. E. O'Malley, Jr. in C. Dunkl, M. Ismail, R. Wong (eds.). *Special Functions*, World Scientific, Singapore, 2000. *BookLink*

★1034 W. Opechowski. *Crystallographic and Metacrystallographic Groups*, North-Holland, Amsterdam, 1986. *BookLink*

★1035 Y. Oono. *Int. J. Mod. Phys.* B 14, 1327 (2000). *DOI-Link*

★1036 A. C. Or. *SIAM J. Appl. Math.* 54, 597 (1994).

★1037 T. Ord. *arXiv:math.LO*/0209332 (2002). *Get Preprint*

★1038 S. Otterbein. *Arch. Rat. Mech. Anal.* 78, 381 (1982).

★1039 J. O'Rourke. *Sigact News* 30, n3, 35 (1999). *DOI-Link*

★1040 M. Pakdemirli, C. Alacaci. *Int. J. Math. Edu. Sci. Technol.* 24, 121 (1993).

★1041 S. Pakvasa, W. Simmons, X. Tata. *arXiv:quant-ph*/9911091 (1999). *Get Preprint*

★1042 B. S. Palmer. *Eur. J. Phys.* 25, 655 (2004). *DOI-Link*

★1043 M. Papadopoulos. *Proc. R. Soc. Lond.* A 273, 198 (1963).

★1044 J. G. Papastavridis. *Tensor Calculus and Analytical Dynamics*, CRC Press, Boca Raton, 1999. *BookLink*

★1045 J. G. Papastavridis. *Analytical Mechanics: A Comprehensive Treatise of the Dynamics of Constrained Systems; for Physicists and Mathematicians*, Oxford University Press, Oxford, 2002. *BookLink*

★1046 G. C. Paquette. *Physica* A 276, 122 (2000). *DOI-Link*

★1047 J. Paradís, L. Bibiloni, P. Viader. *Order* 13, 369 (1996).

★1048 R. B. Paris, D. Kamisnki. *Asymptotics and the Mellin–Barnes Integrals*, Cambridge University Press, Cambridge, 2001. *BookLink*

★1049 P. C. Paris, L. Zhang. *Math. Notes* 36, 855 (2002).

★1050 D. K. Park. *J. Phys.* A 29, 6407 (1996). *DOI-Link*

★1051 R.G. Parr, W. Yang. *Density Functional Theory of Atoms and Molecules*, Oxford University Press, Oxford, 1989. *BookLink (2)*

★1052 J. M. R. Parrando, G. P. Harmer, D. Abbott. *Phys. Rev. Lett.* 85, 5226 (2000). *DOI-Link*

★1053 M. H. Partovi, E. J. Morris. *arXiv:physics*/0406085 (2004). *Get Preprint*

★1054 G. Parzen. *Phys. Rev.* 89, 237 (1953). *DOI-Link*

★1055 R. Paskauškas, L. You. *Phys. Rev.* A 64, 042310 (2001). *DOI-Link*

★1056 A. K. Pati, S. R. Jain, A. Mitra, R. Ramanna. *arXiv:quant-ph*/0207144 (2002). *Get Preprint*

★1057 S. H. Patil. *J. Chem. Phys.* 120, 6399 (2004). *DOI-Link*

★1058 W. H. Paulsen. *Am. Math. Monthly* 101, 953 (1994).

★1059 R. L. Pego, J. R. Quintero. *Physica* D 132, 476 (1999). *DOI-Link*

★1060 A. R. Penner. *Am. J. Phys.* 69, 332, (2001). *DOI-Link*

★1061 R. Penrose in P. Århem, H. Liljenström, U. Svedin (eds.). *Matter Matters?*, Springer, Berlin, 1997. *BookLink*

★1062 A. V. Penskoi. *J. Phys.* A 35, 425 (2002). *DOI-Link*

★1063 U. Pesavento, Z. J. Wang. *Phys. Rev. Lett.* 93, 144501 (2004). *DOI-Link*

★1064 O. Peters, C. Hertlein, K. Christensen. *Phys. Rev. Lett.* 88, 018701 (2002). *DOI-Link*

★1065 O. Peters, K. Christensen. *arXiv:cond-mat*/0204109 (2002). *Get Preprint*

★1066 M. Petkovsek, H. S. Wilf, D. Zeilberger. *A+B*, A K Peters, Wellesley, 1996. *BookLink*

★1067 D. Petrie, J. L. Hunt, C. G. Gray. *Am. J. Phys.* 70, 1025 (2002). *DOI-Link*

★1068 E. Petrisor. *Physica* D 112, 319 (1998). *DOI-Link*

★1069 F. Pfeiffer, C. Glocker. *Multibody Dynamics with Unilateral Contacts*, Wiley, New York, 1996. *BookLink (2)*

★1070 J. Piasecki, C. Gruber. *arXiv:cond-mat*/9810196 (1998). *Get Preprint*

★1071 R. Picard. *Ricerchi. Matematica* 47, 153 (1998).

★1072 D. Picca. *J. Phys.* A 15, 2801 (1982). *DOI-Link*

★1073 E. Piña, T. Ortiz. *J. Phys.* A 21, 1293 (1988). *DOI-Link*

★1074 H. A. Pinnow, K. J. Wiese. *arXiv:cond-mat*/0110011 (2001). *Get Preprint*

★1075 M. A. Pinsky. *Notices Am. Math. Soc.* 42, 330 (1995).

★1076 M. A. Pinsky. *Commun. Pure Appl. Math.* 47, 653 (1994).

★1077 M. A. Pinsky. *Expos. Math.* 18, 357 (2000).

★1078 I. Pitowsky. *Quantum Probability-Quantum Logic*, Springer-Verlag, Berlin, 1989. *BookLink*

✶1079 A. O. Pittenger, M. H. Rubin. *arXiv:quant-ph*/0308142 (2003). *Get Preprint*

✶1080 D. Place, P. Villedieu. *J. Comput. Phys.* 150, 332 (1999). *DOI-Link*

✶1081 M. V. Pletyukhov, E. A. Tolkachev. *J. Math. Phys.* 40, 93 (1999). *DOI-Link*

✶1082 W. A. Pliskin. *Am. J. Phys.* 34, 28 (1960).

✶1083 M. S. Plyushchay, M. R. de Traubenberg. *arXiv:hep-th*/0001067 (2000). *Get Preprint*

✶1084 H. Pollard. *Am. Math. Monthly* 79, 495 (1972).

✶1085 B. Polster. *The Mathematics of Juggling*, Springer-Verlag, New York, 2002. *BookLink*

✶1086 V. T. Portman. *Comput. Meth. Appl. Mech. Eng.* 135, 63 (1996).

✶1087 E. A. Power, T. Thirunamachandran. *Proc. R. Soc. Lond.* A 313, 403 (1969).

✶1088 E. A. Power, T. Thirunamachandran. *Am. J. Phys.* 70, 1136 (2002). *DOI-Link*

✶1089 C. Pozrikidis. *Eng. Anal. Bound. Elem.* 28, 315 (2004). *DOI-Link*

✶1090 V. V. Prasalov. *Polynomials*, Springer, Berlin, 2004. *BookLink*

✶1091 J.-P. Provost in D. Benest, C. Froeschle (eds.). *An Introduction to Methods of Complex Analysis and Geometry for Classical Mechanics and Nonlinear Waves*, Frontieres, France, 1994. *BookLink*

✶1092 D. Prato, R. J. Gleiser. *Am. J. Phys.* 50, 536 (1982). *DOI-Link*

✶1093 W. H. Press, S. A. Teukolsky, W. T. Vetterling, B. P. Flannery. *Numerical Recipes in C*, Cambridge University Press, Cambridge, 1992. *BookLink (4)*

✶1094 R. H. Price, J. D. Romano. *Am. J. Phys.* 66, 109 (1998). *DOI-Link*

✶1095 T. Prosen, D. K. Campbell. *arXiv:chao-dyn*/9908021 (1999). *Get Preprint*

✶1096 G. Pruessner, H. J. Jensen. *arXiv:cond-mat*/0309173 (2003). *Get Preprint*

✶1097 V. I. Pupyshev, A. Y. Ermilov. *Int. J. Quant. Chem.* 96, 185 (2004). *DOI-Link*

✶1098 F. Puel. *Celest. Mech. Dynam. Astron.* 74, 199 (1999). *DOI-Link*

✶1099 Z. Qian, V. Sahni. *Phys. Lett.* A 248, 393 (1998). *DOI-Link*

✶1100 W. Qiao. *J. Phys.* A 29, 2257 (1996). *DOI-Link*

✶1101 C. Quigg. *arXiv:hep-ph*/0502070 (2005). *Get Preprint*

✶1102 P. V. Quinn, Sr., D. C. Hong, J. A. Both. *arXiv:cond-mat*/0409424 (2004). *Get Preprint*

✶1103 A. Rababah. *Proc. Am. Math. Soc.* 119, 803 (1993).

⋆1104 A. Rababah. *Comput. Aided Geom. Design* 12, 89 (1995). *DOI-Link*

⋆1105 P. J. Rabier, W. C. Rheinholdt. *Nonholonomic Motion of Rigid Mechanical Systems From a DAE Viewpoint*, SIAM, Philadelphia, 2000. *BookLink*

⋆1106 P. Rabinowitz. *J. ACM* 13, 296 (1966). *DOI-Link*

⋆1107 D. A. Rabson, J. F. Huesman, B. N. Fisher. *Found. Phys.* 33, 1769 (2003). *DOI-Link*

⋆1108 Q. I. Rahman, G. Schmeisser. *Analytic Theory of Polynomials*, Oxford University Press, Oxford, 2002. *BookLink*

⋆1109 S. G. Rajeev. *arXiv:hep-th*/0210179 (2002). *Get Preprint*

⋆1110 B. Rajagopalan, P. G. Tarboton in T. Vicsek, M. Shlesinger, M. Matsushita (eds.). *Fractals in Natural Sciences*, World Scientific, Singapore, 1994. *BookLink*

⋆1111 K. Ramasubramanian, M. S. Sriram. *arXiv:chao-dyn*/9909029 (1999). *Get Preprint*

⋆1112 L. Rasmusson, M. Boman. *arXiv:physics*/0210094 (2002). *Get Preprint*

⋆1113 R. T. Rau, D. Weiskopf, H. Ruder in H.-C. Hege, K. Polthier (eds.). *Mathematical Visualization*, Springer-Verlag, Heidelberg, 1998. . *BookLink*

⋆1114 J. R. Ray. *Am. J. Phys.* 34, 406 (1966).

⋆1115 G. Rayna. *Reduce: Software for Algebraic Computation*, Springer-Verlag, Berlin, 1987. *BookLink*

⋆1116 D. Redfern. *The Maple Handbook*, Springer-Verlag, New York, 1993. *BookLink (2)*

⋆1117 D. A. Redelmeier, R. J. Tibshirani. *Chance* 13, n13, 8 (2000).

⋆1118 W. J. Reed, B. D. Hughes. *Physica* A 319, 579 (2003). *DOI-Link*

⋆1119 L. Reich in S. D. Chatterji, B. Fuchssteiner, U. Kulisch, D. Laugwitz, R. Liedl (eds.) *Jahrbuch Überblicke Mathematik 1979*, BI, Mannheim, 1979.

⋆1120 W. P. Reid. *Am. J. Phys.* 31, 565 (1963).

⋆1121 W. P. Reid. *SIAM J. Appl. Math.* 15, 1 (1967).

⋆1122 M. Reiher, B. Heß in J. Grotendorst (ed.). *Modern Methods and Algorithms of Quantum Chemistry* John von Neumann Institut for Computing, Jülich, 2000. http://www.kfa-juelich.de/nic-series/Volume3/Volume3.htm

⋆1123 M. Reiher, A. Wolf. *J. Phys. Chem.* 121, 2037 (2004). *DOI-Link*

⋆1124 M. Reiher, A. Wolf. *J. Chem. Phys.* 121, 10945 (2004). *DOI-Link*

⋆1125 J. M. Renes, R. Blume–Kohout, A. J. Scott, C. M. Caves. *arXiv:quant-ph*/0310075 (2003). *Get Preprint*

⋆1126 L. Renna in M. Boiti, L. Martina, F. Pempinelli, B. Prinari, G. Soliani (eds.). *Nonlinearity, Integrability and all that: Twenty Years after NEEDS'79*, World Scientific, Singapore, 2000. *BookLink*

✱1127 L. Renna. *Phys. Rev.* E 64, 046213 (2001). *DOI-Link*

✱1128 A. Rényi. *Acta Math.* 8, 477 (1957).

✱1129 R. Resch, F. Stenger, J. Waldvogel. *Aequ. Math.* 60, 25 (2000). *DOI-Link*

✱1130 L. R. Ribeiro, C. Furtado, J. R. Nascimento. *arXiv:quant-ph*/0502129 (2005). *Get Preprint*

✱1131 N. W. Rickert. *Am. Math. Monthly* 75, 166 (1968).

✱1132 F. Ritort. *Sem. Poincaré* 2, 63 (2003).

✱1133 I. Rodriguez-Iturbe, D. R. Cox, F. R. S. Isham, V. Isham. *Proc. R. Soc. Lond.* A 410, 269 (1987).

✱1134 I. Rodriguez-Iturbe, D. R. Cox, F. R. S. Isham, V. Isham. *Proc. R. Soc. Lond.* A 417, 283 (1988).

✱1135 I. Rodriguez-Iturbe, A. Rinaldo. *Fractal River Basins*, Cambridge University Press, Cambridge, 1997. *BookLink (2)*

✱1136 R. L. Ricca. *Banach Center Publ.* 42, 321 (1998).

✱1137 J. R. Rice. *Numerical Methods, Software and Analysis*, Academic Press, Boston, 1993. *BookLink (3)*

✱1138 D. Richards. *Advanced Mathematical Methods with Maple*, Cambridge University Press, Cambridge, 2002. *BookLink (2)*

✱1139 T. M. Richardson. *arXiv:math.LA*/9905079 (1999). *Get Preprint*

✱1140 S. W. Rienstra. *J. Eng. Math.* 24, 193 (1990).

✱1141 J. Roberts. *Lure of Integers*, American Mathematical Society, 1992. *BookLink*

✱1142 R. W. Robinett. *Am. J. Phys.* 65, 1167 (1997). *DOI-Link*

✱1143 R. W. Robinett. *Am. J. Phys.* 67, 67 (1999). *DOI-Link*

✱1144 R. W. Robinett. *J. Math. Phys.* 40, 101 (1999). *DOI-Link*

✱1145 C. Rorres. *Math. Intell.* 26, n3, 32 (2004).

✱1146 F. Roesler. *Arch. Math.* 73, 193 (1999). *DOI-Link*

✱1147 R. R. J. Rohr. *Die Sonnenuhr*, Callwey, München, 1982. *BookLink*

✱1148 P. Rosenau, A. Oron, J. M. Hyman. *Phys. Fluids* A 4, 1102 (1992). *DOI-Link*

✱1149 S. Rosswog, P. Wagner. *arXiv:cond-mat*/0110101 (2001). *Get Preprint*

✱1150 H. C. Rosu. *Mod. Phys. Lett.* A 18, 1205 (2003). *DOI-Link*

✱1151 H. C. Rosu, M. Planat, M. Saniga. *arXiv:quant-ph*/0409096 (2004). *Get Preprint*

✱1152 B. F. Rothenstein, C. Tamasdan. *Eur. J. Phys.* 15, 16 (1994). *DOI-Link*

✱1153 B. Rotman. *Phil. Trans. R. Soc. Lond.* A 361, 1675 (2003). *DOI-Link*

✱1154 P. Roura, J. Fort, J. Saurina. *Eur. J. Phys.* 21, 95 (2000). *DOI-Link*

✱1155 J. Rudnick, G. Gaspari. *Elements of the Random Walk*, Cambridge University Press, Cambridge, 2004. *BookLink*

✱1156 P. Saari, M. Menert, H. Valtna. *arXiv:quant-ph*/0409034 (2004). *Get Preprint*

✱1157 J. Sakhr, N. D. Whelan. *arXiv:nlin.CD*/0001051 (2000). *Get Preprint*

✱1158 A. Salat. *Z. Naturf.* a 40, 959 (1985).

✱1159 E. J. Saletan, A. H. Cromer. *Am. J. Phys.* 38, 892 (1970).

✱1160 E. Salkowski. *Sitzungsber. Berlin. Math. Ges.* 10, 23 (1910).

✱1161 L. I. Salminen, A. I. Tolvanen, M. J. Alava. *Phys. Rev. Lett.* 89, 185503 (2002). *DOI-Link*

✱1162 L. I. Salminen, A. I. Tolvanen, M. J. Alava. *arXiv:cond-mat*/0301299 (2003). *Get Preprint*

✱1163 S. G. Samko, A. A. Kilbas, O. I. Marichev. *Fractional Integrals and Derivatives*, Gordon and Breach, New York, 1993. *BookLink*

✱1164 D. P. Sanders. *arXiv:nlin.CD*/0411012 (2004). *Get Preprint*

✱1165 J. R. Sanmartin, M. A. Vallejo. *Am. J. Phys.* 46, 949 (1978). *DOI-Link*

✱1166 G. Sansone. *Orthogonal Functions*, Interscience Publishers, New York, 1959. *BookLink (3)*

✱1167 K. Sasaki. *arXiv:physics*/0310163 (2003). *Get Preprint*

✱1168 A. Schadschneider. *arXiv:cond-mat* 9711296 (1997). *Get Preprint*

✱1169 A. Schadschneider. *arXiv:cond-mat*/9902170 (1999). *Get Preprint*

✱1170 A. Schadschneider. *Physica* A 285, 101 (2000). *DOI-Link*

✱1171 A. Schadschneider. *arXiv:cond-mat*/0112117 (2001). *Get Preprint*

✱1172 A. Schadschneider. *Physica* A 313, 153 (2002). *DOI-Link*

✱1173 W. L. Schaich. *Phys. Rev.* E 64, 046605 (2001). *DOI-Link*

✱1174 W. L. Schaich. *Am. J. Phys.* 69, 1267 (2001). *DOI-Link*

✱1175 W. Schaub. *Die Sterne* 203 (1957).

✱1176 G. Scheffers. *Sitzungsber. Berlin. Math. Ges.* 8, 122 (1909).

✶1177 K. Schenk, B. Drossel, F. Schwabl. *arXiv:cond-mat*/0105121 (2001). *Get Preprint*

✶1178 K. Scherer in M. W. Müller, M. Felten, D. H. Mache (eds.). *Approximation Theory*, Akademie Verlag, Berlin 1995. *BookLink*

✶1179 A. Schindlmayr. *arXiv:physics*/9903021 (1999). *Get Preprint*

✶1180 R. Schinzinger, P.A.A. Laura. *Conformal Mapping: Methods and Applications*, Elsevier, Amsterdam, 1991. *BookLink (2)*

✶1181 L. Schlesinger. *Math. Z.* 33, 33 (1931).

✶1182 M. Schlosshauer, A. Fine. *arXiv:quant-ph*/0312086 (2003). *Get Preprint*

✶1183 M. Schmick, M. Markus. *Phys. Rev.* E 70, 065101 (2004). *DOI-Link*

✶1184 A. G. M. Schmidt, B. K. Cheng, M. G. E. da Luz. *arXiv:quant-ph*/0211193 (2002). *Get Preprint*

✶1185 E. Schmidt. *Math. Ann.* 63, 433 (1907).

✶1186 T. Schmidt, M. Marhl. *Eur. J. Phys.* 18, 377 (1997). *DOI-Link*

✶1187 M. Schreckenberg in A. Beutelspacher, N. Henze, U. Kulisch, H. Wußing (eds.). *Überblicke Mathematik*, *1998*, Vieweg, Braunschweig, 1998. *BookLink*

✶1188 H. Schumacher. *Sonnenuhren*, Callwey, München, 1973. *BookLink*

✶1189 V. Schwämmle, H. J. Herrmann. *arXiv:cond-mat*/0301589 (2003). *Get Preprint*

✶1190 R. L. E. Schwarzenberger. *Proc. Cambr. Phil. Soc.* 72, 325 (1972).

✶1191 R. L. E. Schwarzenberger. *Proc. Cambr. Phil. Soc.* 76, 23 (1974).

✶1192 J. Schwinger, L. L. DeRaad, Jr., K. A. Milton, W.-Y. Tsai. *Classical Electrodynamics*, Perseus, Reading, 1998. *BookLink*

✶1193 A. Sciarrino. *arXiv:math-ph*/0102022 (2001). *Get Preprint*

✶1194 A. Sciarrino. *arXiv:math-ph*/0111006 (2001). *Get Preprint*

✶1195 P. Šeba, U. Kuhl, M. Barth, H.-J. Stöckmann. *J. Phys.* A 32, 8225 (1999). *DOI-Link*

✶1196 D. M. Sedrakian, A. Z. Khachatrian. *Ann. Phys.* 11, 503 (2002). *DOI-Link*

✶1197 Z. F. Seidov, P. I. Skvirsky. *arXiv:astro-ph*/0002496 (2000). *Get Preprint*

✶1198 P. Serra, S. Kais. *Phys. Rev. Lett.* 77, 466 (1996). *DOI-Link*

✶1199 J. P. Sethna, K. A. Dahmen, C. R. Myers. *arXiv:cond-mat*/0102091 (2001). *Get Preprint*

✶1200 R. U. Sexl, H. Urbantke. *Relativität-Gruppen-Teilchen*, Springer-Verlag, Wien, 1976. *BookLink*

★1201 M. R. A. Shegelski, R. Niebergall, M. A. Walton. *Can. J. Phys.* 74, 663 (1996).

★1202 M. R. A. Shegelski, R. Niebergall. *Austral. J. Phys.* 52, 1025 (1999).

★1203 M. R. A. Shegelski, M. Reid. *Can. J. Phys.* 77, 903 (2000). *DOI-Link*

★1204 M. R. A. Shegelski, M. Reid, R. Niebergall. *Can. J. Phys.* 77, 903 (2000). *DOI-Link*

★1205 D. Shelupsky. *Am. Math. Monthly* 87, 210 (1980).

★1206 H.-M. Shen, T. T. Wu. *J. Math. Phys.* 30, 2721 (1989). *DOI-Link*

★1207 T. Shigehara, H. Mizoguchi, T. Mishima, T. Cheon. *arXiv:quant-ph*/9812006 (1998). *Get Preprint*

★1208 T. Shigehara, H. Mizoguchi, T. Mishima, T. Cheon. *arXiv:quant-ph*/9911059 (1999). *Get Preprint*

★1209 T. Shigehara, H. Mizoguchi, T. Mishima, T. Cheon. *arXiv:quant-ph*/9912049 (1999). *Get Preprint*

★1210 S. Shnider, P. Winternitz. *J. Math. Phys.* 25, 3155 (1984). *DOI-Link*

★1211 A. Sihvola. *IEEE Trans. Antennas Prop.* 52. 2226 (2004).

★1212 M. D. Simon, L. O. Heflinger, S. L. Ridgway. *Am. J. Phys.* 65, 286 (1997). *DOI-Link*

★1213 M. D. Simon, L. O. Heflinger, A. K. Geim. *Am. J. Phys.* 69, 702 (2001). *DOI-Link*

★1214 A. Sisman, H. Saygin. *J. Phys.* D 32, 664 (1999). *DOI-Link*

★1215 A. Sisman, H. Saygin. *Appl. Energy* 68, 367 (2001). *DOI-Link*

★1216 A. Sisman, H. Saygin. *J. Appl. Phys.* 90, 3086 (2001). *DOI-Link*

★1217 D. Sklavenites. *Am. J. Phys.* 65, 225 (1997). *DOI-Link*

★1218 P. F. Slade. *J. Math. Biol.* 42, 41 (2001).

★1219 J. Slepian. *Am. J. Phys.* 19, 87 (1951).

★1220 W. R. Smythe. *Static and Dynamic Electricity*, McGraw-Hill, New York, 1968. *BookLink*

★1221 E. Sober. *Synthese* 115, 355 (1998). *DOI-Link*

★1222 E. Sober, M. Steel. *J. Theor. Biol.* 218, 395 (2002). *DOI-Link*

★1223 F. Soddy. *Nature* 137, 1021 (1936).

★1224 B. Söderberg. *Phys. Rev.* A 46, 1859 (1992). *DOI-Link*

★1225 D. Solli, R. Y. Chia, J. M. Hickmann. *Phys. Rev.* E 66, 056601 (2002). *DOI-Link*

★1226 F. Sols, M. Macucci. *Phys. Rev.* B 41, 11887 (1990). *DOI-Link*

★1227 J. Sondow. *Proc. Am. Math. Soc.* 126, 1311 (1996). *DOI-Link*

★1228 H. Soodak. *Am. J. Phys.* 70, 815 (2002). *DOI-Link*

★1229 S. Sorgatz, S. Wehmeier. *Math. Comput. Simul.* 49, 235 (1999). *DOI-Link*

★1230 B. Souvignier. *Acta Cryst.* A 59, 210 (2003). *DOI-Link*

★1231 Special interest group of GI, DMV, GAMM, (1991). http://www.uni-karlsruhe.de/~CAIS/

★1232 Special Issue on OpenMath. *SIGSAM Bull.* 34 (2000).

★1233 K. J. Spyrou, J. M. T. Thompson. *Phil. Trans. R. Soc. Lond.* A 358, 1733 (2000). *DOI-Link*

★1234 R. Srikanth. *quant-ph*/0302160 (2003). *Get Preprint*

★1235 V. K. Srinivasan. *Int. J. Math. Edu. Sci. Technol.* 28, 185 (1997).

★1236 H. M. Srivastava, R. K. Saxena. *Appl. Math. Comput.* 118, 1 (2001). *DOI-Link*

★1237 J. D. Stadler. *Discr. Math.* 258, 179 (2002). *DOI-Link*

★1238 A. A. Stanislavsky, K. Weron. *Physica* D 156, 247 (2001). *DOI-Link*

★1239 R. P. Stanley. *arXiv:math.CO*/0501256 (2005). *Get Preprint*

★1240 D. Stauffer. *Int. J. Mod. Phys.* C 7, 759 (1996).

★1241 W.-H. Steeb, D. Lewien. *Algorithms and Computation with REDUCE*, BI-Verlag, Mannheim, 1992. *BookLink*

★1242 W. Steeb. *Quantum Mechanics Using Computer Algebra*, World Scientific, Singapore, 1994. *BookLink*

★1243 M. Steel, A. McKenzie in M. Lässig, A. Valleriani (eds.). *Biological Evolution and Statistical Physics*, Springer-Verlag, Berlin, 2002. *BookLink*

★1244 J. M. Steele. *The Cauchy–Schwarz Master Class*, Cambridge University Press, Cambridge, 2004. *BookLink (2)*

★1245 W. Steiner, H. Troger. *ZAMP* 46, 960 (1995).

★1246 S. Stenlund. *Combinators, λ-Terms and Proof Theory*, Reidel, Dordrecht, Holland, 1972. *BookLink*

★1247 R. Stephan. *arXiv:math.CO*/0409509 (2004). *Get Preprint*

★1248 I. Stewart, M. Golubitsky. *Fearful Symmetry*, Blackwell, Oxford, 1992. *BookLink (2)*

★1249 J. Stillwell. *Am. Math. Monthly* 108, 70 (2001).

★1250 O. Stock in S. A. Cerri, G. Gouardères, F. Paraguaçu (eds.). *Intelligent Tutoring Systems*, Springer-Verlag, Berlin, 2002. *BookLink*

★1251 R. G. Stoneham. *Acta Arithm.* 22, 371 (1973).

★1252 R. G. Stoneham. *Acta Arithm.* 42, 265 (1983).

★1253 R. Strebel. *Elem. Math.* 58, 141 (2003). *DOI-Link*

★1254 W. J. Stronge. *Impact Mechanics*, Cambridge University Press, Cambridge, 2000. *BookLink (2)*

★1255 E. Study. *Math. Annalen* 49, 497 (1897).

★1256 A. S. Sumbatov. *Reg. Chaotic Dynam.* 7, 221 (2002). *DOI-Link*

★1257 D. B. Summer. *Trans. Am. Math. Soc.* 87, 526 (1958).

★1258 H. C. Sun, D. N. Metaxas. *Comput. Graphics Proc. SIGGRAPH 2001* 261 (2001). *DOI-Link*

★1259 D. Szász (ed.). *Hard Ball Systems and the Lorentz Gas*, Springer-Verlag, Berlin, 2000. *BookLink*

★1260 B. Tabarrok, F. P. J. Rimrott. *Variational Methods and Complementary Formulations in Dynamics*, Kluwer, Dordrecht, 1994. *BookLink*

★1261 Y. Tajima, T. Nagatani. *Physica* A 292, 545 (2001). *DOI-Link*

★1262 M. Takayasu, K. Fukuda, H. Takayasu. *Physica* A 274, 140 (1999). *DOI-Link*

★1263 Y. Tanabe, K. Kaneko. *Phys. Rev. Lett.* 73, 1372 (1994). *DOI-Link*

★1264 C. Tang. *arXiv:cond-mat*/9912450 (1999). *Get Preprint*

★1265 T. W. Tang, A. Allison, D. Abbott. *arXiv:cs.GT*/0404016 (2004). *Get Preprint*

★1266 S. Tavaré, O, Zeitouni. *Lectures on Probability Theory and Statistics*, Springer-Verlag, New York, 2003. *BookLink*

★1267 B. Y. Tay, M. J. Edirisinghe. *Proc. R. Soc. Lond.* A 458, 2039 (2002). *DOI-Link*

★1268 F. L. Teixeira, W. C. Chew. *J. Math. Phys.* 40, 169 (1999). *DOI-Link*

★1269 V. Ter-Antonyan. *arXiv:quant-ph*/0003106 (2000). *Get Preprint*

★1270 J. Terrel. *Phys. Rev.* 116, 1041 (1959). *DOI-Link*

★1271 J. M. Thijssen. *Computational Physics*, Cambridge University Press, Cambridge, 1999. *BookLink (3)*

★1272 W. J. Thompson. *Am. J. Phys.* 60, 425 (1992). *DOI-Link*

★1273 R. Thompson. *Coll. Math. J.* 29, 48 (1998).

★1274 H. Tietze. *Elem. Math.* 3, 97 (1948).

★1275 D. Tilbury, R. M. Murray, S. S. Sastry. *IEEE Trans. Automat. Contr.* 40, 802 (1995).

✶1276 H. G. Timmer, J. M. Stern. *Comput. Aided Design* 12, 301 (1980). *DOI-Link*

✶1277 I. R. Titze. *Principles of Voice Production*, Prentice-Hall, Englewood Cliffs, 1993. *BookLink*

✶1278 R. Toral. *arXiv:cond-mat*/0101435 (2001). *Get Preprint*

✶1279 G. F. Torres del Castillo. *J. Math. Phys.* 36, 3413 (1995). *DOI-Link*

✶1280 N. Trefethen. *SIAM News* 35, n1, 1 (2002).

✶1281 N. Trefethen. *SIAM News* 35, n6, 1 (2002).

✶1282 M. Trott et al. *SIGSAM Bull.* 31, n4, 2 (1997). *DOI-Link*

✶1283 M. Trott. *The Mathematica GuideBook for Graphics*, Springer-Verlag, New York, 2004. *BookLink*

✶1284 M. Trott. *The Mathematica GuideBook for Numerics*, Springer-Verlag, New York, 2005. *BookLink*

✶1285 M. Trott. *The Mathematica GuideBook for Symbolics*, Springer-Verlag, New York, 2005. *BookLink*

✶1286 C. Tsallis, A. R. Plastino, W.-M. Zheng. *Chaos, Solitons, Fractals* 8, 885 (1997). *DOI-Link*

✶1287 I. Tsutsui, T. Fülöp, T. Cheon. *arXiv:quant-ph*/0003069 (2000). *Get Preprint*

✶1288 I. Tsutsui, T. Fülöp, T. Cheon. *arXiv:math-ph*/0105019 (2001). *Get Preprint*

✶1289 I. Tsutsui, T. Fülöp, T. Cheon. *J. Math. Phys.* 42, 5687 (2001). *DOI-Link*

✶1290 R. Tumulka, N. Zanghì. *arXiv:quant-ph*/0309021 (2003). *Get Preprint*

✶1291 A. Turbiner, P. Winternitz. *Lett. Math. Phys.* 50, 189 (1999). *DOI-Link*

✶1292 J. E. Turner. *Am. J. Phys.* 45, 758 (1977). *DOI-Link*

✶1293 J. Turulski, J. Niedzielski. *J. Math. Chem.* 36, 29 (2004). *DOI-Link*

✶1294 T. Uchino, I. Tsutsui. *arXiv:hep-th*/0302089 (2003). *Get Preprint*

✶1295 F. E. Udwadia, R. E. Kalaba. *Int. J. Nonl. Mech.* 37, 1079 (2002). *DOI-Link*

✶1296 J. Ueberberg. *Einführung in die Computeralgebra mit REDUCE*, BI-Verlag, Mannheim, 1992. *BookLink*

✶1297 S. Ulam in D. Mauldin (ed.). *The Scottish Book*, Birkhäuser, Boston, 1981. *BookLink (2)*

✶1298 D. Ullmo, T. Nagano, S. Tomosovic, H. U. Baranger. *arXiv:cond-mat*/0007330 (2000). *Get Preprint*

✶1299 K. Umeno. *arXiv:chao-dyn*/9812013 (1998). *Get Preprint*

✶1300 M. A. Vandyck. *Eur. J. Phys.* 22, 79 (2001). *DOI-Link*

✶1301 J. Van Bladel. *IEEE Antennas Prop. Mag.* 45, 118 (2003).

✶1302 P. M. van den Berg. *J. Opt. Soc. Am.* 63, 1588 (1973).

✶1303 J. P. van der Weele, E. J. Banning. *Am. J. Phys.* 69, 953 (2001). *DOI-Link*

✶1304 J. P. van der Weele, E. J. Banning. *Nonlinear Phenomena Complex Systems* 3, 268 (2001). http://www.j-npcs.org/abstracts/vol2000/v3no3/v3no3p268.html

✶1305 J. H. G. M. van Geffen, V. V. Meleshko, G. J. F. van Heijst. *Phys. Fluids* 8, 2393 (1996). *DOI-Link*

✶1306 E. van Lenthe, E. J. Baerends, J. G. Snijders. *J. Chem. Phys.* 105, 2373 (1996). *DOI-Link*

✶1307 C. Vanneste. *Eur. J. Phys.* B 23, 391 (2001). *DOI-Link*

✶1308 L. van Wijngaarden. *Theor. Comput. Fluid Dyn.* 10, 449 (1998). *DOI-Link*

✶1309 B. P. van Zyl, D. A. W. Hutchinson. *arXiv:nlin.CD*/0304038 (2003). *Get Preprint*

✶1310 G. Varieschi, K. Kamiya. *arXiv:physics*/0210033 (2002). *Get Preprint*

✶1311 G. L. Vasconcelos, J. J. P. Veerman. *arXiv:cond-mat*/9904139 (1999). *Get Preprint*

✶1312 R. C. Vaughan in A. D. Pollington, W. Moran (eds.). *Number Theory with an Emphasis on the Markov Spec*‐ *trum*, Marcel Dekker, New York, 1993. *BookLink*

✶1313 R. C. Vaughan. *J. Austral. Math. Soc.* 60, 260 (1996).

✶1314 L. Vázquez in F. K. Abdullaev, V. V. Konotop (eds.). *Nonlinear Waves: Classical and Quantum Aspects*, Kluwer, Dordrecht, 2004. *BookLink (2)*

✶1315 V. Vedral. *arXiv:quant-ph*/0302040 (2003). *Get Preprint*

✶1316 D. Velleman, S. Wagon. *Mathematica Edu. Res.* 9, n 3/4, 85 (2001).

✶1317 G. Venezian. *Il. Nuov. Cim.* 111, 1315 (1996).

✶1318 S. T. Venkataraman. *Robot. Auton. Syst.* 22, 75 (1997).

✶1319 F. Vera. *arXiv:nlin.PS*/0206039 (2002). *Get Preprint*

✶1320 J. A. M. Vermaseren. *arXiv:math-ph*/0010025 (2000). *Get Preprint*

✶1321 J. A. M. Vermaseren. *arXiv:hep-ph*/0211297 (2002). *Get Preprint*

✶1322 R. Veysseyre, H. Veysseyre. *Acta Cryst.* A 58, 429 (2002). *DOI-Link*

✶1323 A. Vierkandt. *Monatsh. Math. Phys.* 3, 31 (1892).

✶1324 A. Vierkandt. *Monatsh. Math. Phys.* 3, 97 (1892).

✶1325 R. Vilela Mendes. *J. Phys.* A 27, 8091 (1994). *DOI-Link*

★1326 R. Vilela Mendes. *arXiv:math-ph*/9907001 (1999). *Get Preprint*

★1327 H. Villat. *Lecons sur la Théorie des Tourbillons*, Gauthier–Villars, 1930. *BookLink*

★1328 S. Virmani, M. F. Sacchi, M. B. Plenio, D. Markham.. *Phys. Lett.* A 288, 62, (2001). *DOI-Link*

★1329 M. Visser. *arXiv:gr-qc*/0002011 (2000). *Get Preprint*

★1330 A. Y. Vlasov. *arXiv:quant-ph*/0302064 (2003). *Get Preprint*

★1331 H. von der Mosel. *Ann. Inst. Henri Poincaré* 16, 137 (1999).

★1332 G. E. Volovik. *arXiv:gr-qc*/0004049 (2000). *Get Preprint*

★1333 J. Vrbik. *Am. J. Phys.* 61, 258 (1993). *DOI-Link*

★1334 S. Wagon. *The Banach–Tarski Paradox*, Cambridge University Press, Cambridge, 1985. *BookLink (2)*

★1335 S. Wagon. *Mathematica in Action*, W. H. Freeman, New York, 1991. *BookLink (4)*

★1336 S. Wagon. *The Mathematica Journal* 3, n4, 58 (1993).

★1337 C. Waksjö, S. Rauch-Wojciechowski. *Math. Phys. Anal. Geom.* 6, 301 (2003). *DOI-Link*

★1338 M. Waldschmidt in R. Balakrishnan, K. S. Padmanabhan, V. Thangaraj (eds.). *Ramanujan Centennial Interna-tional Conference*, Ramanujan Math. Soc., 1988.

★1339 J. Waldvogel. *ZAMP* 27, 867 (1976).

★1340 J. Waldvogel. *ZAMP* 30, 388 (1979).

★1341 J. L. Walsh. *Am. Math. Monthly* 68, 978 (1961).

★1342 R. Walter. *Geom. Dedicata* 27, 219 (1988).

★1343 K. K. Wan, C. Trueman, J. Bradshaw. *Int. J. Theor. Phys.* 39, 127 (2000). *DOI-Link*

★1344 J. Wang, T. L. Beck. *arXiv:cond-mat*/9905422 (1999). *Get Preprint*

★1345 J.-P. Wang. *Kodai Math. J.* 27, 144 (2004).

★1346 X. C. Wang, S. K. Ghosh. *Advanced Theories of Hypoid Gears*, Elsevier, Amsterdam, 1994. *BookLink*

★1347 G. H. Wannier. *J. Math. Phys.* 19, 131 (1978). *DOI-Link*

★1348 K. Watanabe, R. Petit, and M. Neviere. *J. Opt. Soc. Am.* A 19, 325 (2003).

★1349 K. Watanabe. *Radio Sc.* 38, n 2, 2 (2003).

★1350 D. S. Watkins. *SIAM Rev.* 47, 3 (2005). *DOI-Link*

★1351 A. E. Waugh. *Sundials: Their Theory and Construction*, Dover, New York, 1973. *BookLink*

✱1352 I. Webman, J. L. Gruver, S. Havlin. *arXiv:cond-mat*/9904148 (1999). *Get Preprint*

✱1353 F. Wegner. *arXiv:physics*/0203061 (2002). *Get Preprint*

✱1354 F. Wegner. *arXiv:physics*/0205059 (2002). *Get Preprint*

✱1355 K. Weibert, J. Main, G. Wunner. *arXiv:nlin.CD*/0005045 (2000). *Get Preprint*

✱1356 S. Weigert. *arXiv:quant-ph*/0407132 (2004). *Get Preprint*

✱1357 M. Weigt, A. K. Hartmann. *arXiv:cond-mat*/0001137 (2000). *Get Preprint*

✱1358 M. Weigt, A. K. Hartmann. *arXiv:cond-mat*/0009417 (2000). *Get Preprint*

✱1359 K. Weihrauch. *Computable Analysis*, Springer-Verlag, Berlin, 2000. *BookLink*

✱1360 D. Weiskopf, U. Kraus, H. Ruder. *ACM Trans. Graphics* 18, 278 (1999). *DOI-Link*

✱1361 D. Weiskopf, U. Kraus, H. Ruder. *J. Visual. Comput. Anim.* 11, 185 (2001). *DOI-Link*

✱1362 C. Weiss, M. Holthaus. *arXiv:cond-mat*/0206023 (2002). *Get Preprint*

✱1363 E. Weisstein. *CRC Concise Encyclopedia of Mathematics*, CRC, Boca Raton, 1998. *BookLink*

✱1364 R. A. Werner. *Celest. Mech. Dynam. Astron.* 59, 253 (1994).

✱1365 R. F. Werner, M. M. Wolf. *arXiv:quant-ph*/0107093 (2001). *Get Preprint*

✱1366 G. B. West, V. M. Savage, J. Gillooly, B. J. Enquist, W. H. Woodruff, J. H. Brown. *arXiv:physics*/0211058 (2002). *Get Preprint*

✱1367 J. A. White, F. L. Román, A. Gonzáles, S. Velasco. *Europhys. Lett.* 59, 479 (2002). *DOI-Link*

✱1368 E. J. W. Whittaker. *An Atlas of Hyperstereograms of the Four-Dimensional Crystal Classes*, Clarendon Press, Oxford, 1971. *BookLink*

✱1369 R. Wiedemann. *Schriftenreihe des Institutes für Verkehrswesen der Universität Karlsruhe*, n8 (1974).

✱1370 C. Wilcox. *J. Anal. Math.* 33, 146 (1978).

✱1371 J. B. Wilker. *J. Geom.* 55, 174 (1996).

✱1372 A. Willers. *Z. Math. Phys.* 57, 158 (1909).

✱1373 S. W. Williams. *Math. Intell.* 24, n3, 17 (2002).

✱1374 W. Willinger, V. Paxson. *Notices Am. Math. Soc.* 45, 961 (1998).

✱1375 J. Winicour. *arXiv:gr-qc*/0003029 (2000). *Get Preprint*

✱1376 P. Winternitz. *J. Math. Phys.* 25, 2149 (1984). *DOI-Link*

✶1377 O. Winther, A. Krogh. *arXiv:cond-mat*/0307497 (2003). *Get Preprint*

✶1378 D. A. Wisniacki, E. Vergini, R. M. Benito, F. Borondo. *arXiv:nlin.CD*/0311052 (2003). *Get Preprint*

✶1379 P. Wocjan, T. Beth. *arXiv:quant-ph*/0407081 (2004). *Get Preprint*

✶1380 D. E. Wolf, M. Schreckenberg, A. Bachem (eds.). *Traffic and Granular Flow*, World Scientific, Singapore, 1996. *BookLink (4)*

✶1381 M. Wolf. *Physica* A 274, 149 (1999). *DOI-Link*

✶1382 S. Wolfram. *The Mathematica Book*, Cambridge University Press and Wolfram Media, 1999. *BookLink*

✶1383 H. Wondratschek, R. Bülow, J. Neubüser. *Acta Cryst.* A 27, 523 (1971).

✶1384 W. Wong, L. Lee, K. Wong. *Comput. Phys. Commun.* 138, 234 (2001). *DOI-Link*

✶1385 A. J. Wood (ed.). *Physica* A 313, 83 (2002). *DOI-Link*

✶1386 N. M. J. Woodhouse. *Special Relativity*, Springer-Verlag, New York, 1992. *BookLink (3)*

✶1387 S. C. Woon. *Rev. Math. Phys.* 11, 463 (1999). *DOI-Link*

✶1388 W. K. Wootters. *Found. Phys.* 16, 391 (1986).

✶1389 W. K. Wootters. *Ann. Phys.* 176, 1 (1987). *DOI-Link*

✶1390 W. K. Wootters, B. D. Fields. *Ann. Phys.* 191, 363 (1989). *DOI-Link*

✶1391 W. K. Wootters. *arXiv:quant-ph*/0406032 (2004). *Get Preprint*

✶1392 D. R. Wu, J. S. Luo. *A Geometric Theory of Conjugate Tooth Surfaces*, World Scientific, Singapore, 1992. *BookLink*

✶1393 D. W. Wu, N. Baeth. *Int. J. Math. Edu. Sci. Technol.* 32, 774 (2001). *DOI-Link*

✶1394 H. Wu, D. W. L. Sprung. *Phys. Rev.* E 48, 2595 (1993). *DOI-Link*

✶1395 M. Wu, M. Gharib. *arXiv:patt-sol*/9804002 (1998). *Get Preprint*

✶1396 T. T. Wu, M. L. Yu. *J. Math. Phys.* 43, 5949 (2002). *DOI-Link*

✶1397 H.-J. Xu, L. Knopoff. *Phys. Rev.* E 50, 3577 (1994). *DOI-Link*

✶1398 K. Xu, W. Li. *arXiv:cs.AI*/0004005 (2000). *Get Preprint*

✶1399 K. Xu, W. Li. *arXiv:cs.AI*/0005024 (2000). *Get Preprint*

✶1400 Y. Y. Yamaguchi, Y. Nambu. *arXiv:chao-dyn*/9902013 (1999). *Get Preprint*

✶1401 S.-l. Yang. *Discr. Appl. Math.* 146, 102 (2005). *DOI-Link*

✶1402 Y. Yavin. *Math. Comput. Model.* 38, 1029 (2003). *DOI-Link*

✶1403 Z. Ye. *Phys. Lett.* A 327, 91 (2004). *DOI-Link*

✶1404 D. N. Yetter. *arXiv:math.MG*/9809007 (1998). *Get Preprint*

✶1405 H. Yizhaq, N. J. Balmforth, A. Provenzale. *Physica* D 195, 207 (2004). *DOI-Link*

✶1406 Z. Yoshida. *J. Math. Phys.* 33, 1252 (1992). *DOI-Link*

✶1407 D. H. Zanette, S. C. Manrubia. *arXiv:nlin.AO*/0009046 (2000). *Get Preprint*

✶1408 P. Zavada. *Commun. Math. Phys.* 192, 261 (1998). *DOI-Link*

✶1409 D. Zeilberger in D. Stanton (ed.). *q-Series and Partitions*, Springer-Verlag, New York, 1989. *BookLink*

✶1410 D. Zeilberger. *arXiv:math.CO*/9805126 (1998). *Get Preprint*

✶1411 D. Zeilberger. *arXiv:math.CO*/9811070 (1998). *Get Preprint*

✶1412 D. Zeilberger. *Opinions* (1999). http://www.math.temple.edu/~zeilberg/Opinion36.html

✶1413 D. Zeilberger. *Preprint* (2002). http://www.math.rutgers.edu/~zeilberg/mamarim/mamarimhtml/bb.html

✶1414 D. Zeilberger. *Adv. Appl. Math.* 31, 532 (2003). *DOI-Link*

✶1415 P. Zeiner, R. Dirl, B. L. Davies. *J. Math. Phys.* 39, 2437 (1998). *DOI-Link*

✶1416 P. Zeiner, R. Dirl, B. L. Davies. *J. Math. Phys.* 40, 2757 (1999). *DOI-Link*

✶1417 G.-J. Zeng, S.-L. Zhou, S.-M. Ao, F.-S. Jiang. *J. Phys.* A 30, 1775 (1997). *DOI-Link*

✶1418 A. Zenkert. *Faszination Sonnenuhr*, Verlag Technik, Berlin, 1984. *BookLink*

✶1419 Y.-C. Zhang. *arXiv:cond-mat*/0105186 (2001). *Get Preprint*

✶1420 X.-G. Zhao, X.-W. Zahang, S.-G. Chen, W.-X. Zhang. *Int. J. Mod. Phys.* B 4, 4215 (1993). *DOI-Link*

✶1421 Z. Zhang, F. Comellas, G. Fertin, L. Rong. *arXiv:cond-mat*/0503316 (2005). *Get Preprint*

✶1422 A. Zhedanov. *J. Approx. Th.* 97, 1 (1999). *DOI-Link*

✶1423 L. Zhenxiu. *arXiv:math-ph*/0309029 (2003). *Get Preprint*

✶1424 C. Zhu, H. Nakamura. *J. Math. Phys.* 33, 2697 (1992). *DOI-Link*

✶1425 M. Znojil. *arXiv:math-ph*/0002017 (2000). *Get Preprint*

✶1426 M. Znojil, G. Lévai. *arXiv:quant-ph*/0003081 (2000). *Get Preprint*

✶1427 W. H. Zurek. *arXiv:quant-ph*/0211037 (2002). *Get Preprint*

✸1428 W. H. Zurek. *arXiv:quant-ph*/0308163 (2003). *Get Preprint*

✸1429 K. Zyckowski, I. Bengtsson. *Ann. Phys.* 295, 115 (2002). *DOI-Link*

P R O G R A M M I N G

CHAPTER 2

Structure of *Mathematica* Expressions

2.0 Remarks

This chapter starts the systematic discussion of the use of the *Mathematica* programming system and the *Mathematica* language. All *Mathematica* expressions resemble each other because they are symbolic expressions. The whole power, universality, flexibility, and extensibility are based on the unifying fact that everything in *Mathematica* is a symbolic expression. Depending on the size of these symbolic expressions, we can classify them as elementary objects, called atoms, or as objects built recursively from smaller pieces. Elementary objects include strings, symbols, and various types of numbers. More complicated expressions can be decomposed and analyzed using a few basic commands, such as `Level`, `Depth`, `Part`, and `Position`.

Throughout the *GuideBooks*, the author has tried to present *Mathematica* step by step and to make use of functions and programming constructs used in earlier chapters only. However, to provide and discuss some examples, this principle will be relaxed in the first few sections of this chapter.

```
In[1]:= (* no spelling warnings, set fonts for tick labels, ... *)
        Get[ToFileName[ReplacePart["FileName" /.
         NotebookInformation[EvaluationNotebook[]], "Initialization.m", 2]]];
        TMGBs`TMGBsV51::notV51 :
         The inputs of this notebook are tailored for Mathematica 5.1. Some
           inputs might not work properly in earlier versions of Mathematica.
```

2.1 Expressions

All functions, results, syntactically correct inputs and outputs, error messages, and on-line information used in *Mathematica* are expressions. Every expression is formed hierarchically from subexpressions, and every atomic expression has a type. The type of the highest level of an expression is called its *head*. A detailed understanding of the structure of expressions is absolutely essential to understanding the important commands in *Mathematica* that are generally used to manipulate results of larger calculations, graphics, etc. (e.g., `Map`, `Thread`, `MapAt`, `Inner`, `Outer`, `Flatten`, `FlattenAt`, `Distribute`, and `MapThread`). The most important commands for “visually” (meaning by looking at the expression, not carrying out a program on the expression) determining the structure of a simple expression are `FullForm`, `TreeForm`, `InputForm`, and `OutputForm`. For formatted (typeset) input and output, the possible built-in forms are `StandardForm` and `TraditionalForm`.

`FullForm[`*expression*`]`
  gives the internal form of *expression* in the long form of the *Mathematica* functions.

`FullForm` is most convenient for investigating the structure of an expression because no grouping problems exist.

`TreeForm[`*expression*`]`
  gives a hierarchical display of the internal form of *expression* in the long form of the *Mathematica* commands.

Because of the large amount of space required to show a structure in `TreeForm`, it is best used only on smaller expressions. We give explicit examples in the following sections.

A more compact way to view (and input) *Mathematica* expressions is `InputForm`.

`InputForm[`*expression*`]`
  gives the input form of *expression*.

In this form, the long form of many *Mathematica* commands is replaced by a shorter format. As the name suggests, this form represents the one typically used as *Mathematica* input. In `InputForm`, the symbol `*` for multiplication is explicitly displayed. *Mathematica* can return the result of calculations in various forms. A terminal adapted one is `OutputForm`.

`OutputForm[`*expression*`]`
  gives the typical mathematical form of *expression* as formatted by the *Mathematica* front end or in a terminal.

Mathematica input and output can be two-dimensional (2D), meaning it includes growing roots, braces, brackets, fraction bars, summation, and product signs, …. The two forms allowing this type of input and output are `StandardForm` and `TraditionalForm`.

`StandardForm[`*expression*`]`

gives the *Mathematica* form of *expression* as formatted by the *Mathematica* front end using typesetting symbols.

`TraditionalForm[`*expression*`]`

gives the typical mathematical form of *expression* as formatted by the *Mathematica* front end using typesetting symbols.

As mentioned, every expression has a type, called a head.

`Head[`*expression*`]`

gives the head of *expression* (that is the type of the outermost part of an expression).

The most important heads are those of numbers and strings, along with system-defined and user-defined symbols and functions. Here is an example of each type.

The following integer number has the head `Integer`.

```
Head[3]
```

Here is the system function `FullForm` with an argument `x`.

```
Head[FullForm[x]]
```

The head is the symbol `FullForm` (used here for the first time) itself; the head of every elementary user-defined or system-defined symbol is `Symbol`.

```
Head[x]
```

```
Head[Sin]
```

The head of "the function value" `y[3]` (of a function `y` that is not explicitly defined) is `y`.

```
Head[y[3]]
```

The head of the function $y_a(x)$ is y_a. $y_a(x)$ in *Mathematica* is best written as `y[a][x]` (= `(y[a])[x]`). For these kinds of composite expressions, the head is everything except for the last argument(s).

```
Head[y[a][x]]
```

```
Head[y[a][3]]
```

```
Head[y[a][b][c]]
```

For contrast, here is a function `y` with two arguments.

```
Head[y[a, x]]
```

The following expression has the head `y[a][b]`, and its arguments are `w1`, `w2`, and `w3`.

```
Head[y[a][b][w1, w2, w3]]
```

Here is a composition of functions. The function `y[a]` is applied to `b[w1, w2, w3]`.

```
Head[y[a][b[w1, w2, w3]]]
```

If the function takes no argument, as `functionWithNoArguments` in the following example, it nevertheless has a head.

```
Head[functionWithNoArguments[]]
```

Here is the composite head applied to an empty list of arguments.

```
Head[y[3][]]
```

Mathematica expressions can be nested arbitrarily deeply. Here is a more complicated example.

```
a[b[c][d[e[f[g][h[i]]][j[k[l][m[n]]]]]]]
```

The next output is its `TreeForm`.

```
TreeForm[%]
```

For displaying results of *Mathematica* calculations, we will use one of the following four forms. `OutputForm` displays with alignments typically used in a terminal interface. We will use it occasionally for short versions of long, structurally repeating output.

```
OutputForm[Sin[x]^2 + 1/y + α]
```

`StandardForm` displays expressions with square roots, fraction bars, superscripts (for powers), etc. and uses the full names of most *Mathematica* functions, with the exception of the ones having intuitive short cuts used in `InputForm` and a few more. For the vast majority of all calculations, we will use `StandardForm` as the format to return results. Results in `StandardForm` are usually most easy to read. The results returned by *Mathematica* are interactively editable and then again evaluatable.

```
StandardForm[Sin[x]^2 + 1/y + α + Log[ArcSin[Sqrt[z^ξ]]]]
```

`TraditionalForm` displays expressions with square roots, fraction bars, superscripts (for powers), … and uses the names and symbols from traditional mathematics. We will occasionally use it to display particularly nice results. Be aware that the (visible) order of the expressions in a sum is different in `TraditionalForm` than it is in `StandardForm`.

```
TraditionalForm[Sin[x]^2 + 1/y + α + Log[ArcSin[Sqrt[z^ξ]]]]
```

`InputForm` finally uses shortcuts and is a strictly one-dimensional (1D) representation. We will use `InputForm` to format outputs from time to time, especially in cases in which the other three forms `OutputForm`, `StandardForm`, and `TraditionalForm` produce large outputs with a lot of white space.

```
InputForm[Sin[x]^2 + 1/y + α + Log[ArcSin[Sqrt[z^ξ]]]]
```

For the *Mathematica* programs in this book, `InputForm` is best suited because it allows us to align everything and has a constant line height. We will use `InputForm` nearly exclusively throughout the rest of the book for inputs. Also, it can take Greek letters and other special characters. We will make use of Greek and Gothic letters, but we will not use other special characters (such as → or ==). The last sections of Chapter 1 and Chapter 2 of the Graphics volume [66✱] will contain programs that use more abbreviations and symbols.

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

2.2 Simple Expressions

■ 2.2.1 Numbers and Strings

In this subsection, we look carefully at numbers in *Mathematica*. Here is the integer 3 in its `FullForm`, `TreeForm`, `InputForm`, and `OutputForm`.

```
3
```

```
FullForm[3]

TreeForm[3]

InputForm[3]

OutputForm[3]
```

Note that the labels `// FullForm = …` , `// TreeForm = …` , `// InputForm = …` , and `// Output⋱ Form = …` in the output do not belong to the output expressions, but are simply different formattings of the same expression, which (in this case) are all identical. If we recover one of these outputs using `%` or `Out[…]`, the labels are not included, as shown below.

```
%

%%%
```

Here is the head of the number 3.

```
Head[3]
```

Thus, it is an integer (it is at the same time an odd number and a prime, but these properties are not reflected in the head). Negative integers also have the head `Integer`.

```
Head[-3]
```

Next, we look at a rational number. Its `OutputForm` is different from its `FullForm` because the output is formatted as a fraction.

```
FullForm[343/561]

TreeForm[343/561]

InputForm[343/561]

OutputForm[343/561]
```

In `StandardForm`, a fraction also displays with a fraction bar.

```
StandardForm[343/561]
```

The same rule holds for `TraditionalForm`. In `TraditionalForm`, a serif typeface is used.

```
TraditionalForm[343/561]

Head[343/561]
```

Here is a real number that, in *Mathematica*, is a number with a decimal point and a finite number of digits.

```
FullForm[3.987568]

TreeForm[3.987568]

InputForm[3.987568]

OutputForm[3.987568]

Head[3.987568]
```

Here is an exact complex number [54✱]. The imaginary unit is represented by `I` in *Mathematica*. (In `Traditional⋱ Form` *i* is used.)

```
FullForm[3 + 8 I]

OutputForm[3 + 8 I]
```

```
Head[3 + 8 I]
```

Complex numbers with finite accuracy or whose real and imaginary parts are fractions also have the head `Complex`.

```
Head[3.98 + 8.987 I]
```

```
Head[23/17 + 51/89 I]
```

Complex numbers with mixed real and imaginary parts, one being exact and one being approximate, also have the head `Complex`.

```
Head[23/17 + 2.222 I]
```

We now summarize the various number types [56✱].

`Integer`

is the head for a positive or negative integers and 0.

The number 0 is an integer [33✱], [61✱].

```
Head[0]
```

Mathematica automatically simplifies sums and products containing the integers 0 or 1.

```
0 a b c
```

```
0 + a b
```

```
1 u
```

These rules are automatically applied nearly independent of the type of the other summands and factors. Sometimes, these simplifications may result in unexpected results.

```
0 "I am a string"
```

```
0 IAmInfinityBelieveMe + 0 I IAmInfinityTooReally
```

Similarly, the following behavior of *Mathematica* is probably unexpected. Syntactically, this expression is allowed in *Mathematica*, although it does not make much sense semantically.

```
0[0]
```

```
Head[0[0]]
```

`Rational`

is the head for negative and positive rational numbers that do not reduce to an integer.

Integer numbers (head `Integer`) and rational numbers (head `Rational`) are exact, that is, they have no inaccuracy. An exact input to *Mathematica* results in an exact result unless `N` or some numerical routine is used.

The following input does not represent a rational number.

```
-178432511014851389063559176/235465678754467654
```

Indeed, it simplifies to an integer.

```
Head[%]
```

Canceling fractions to a minimal form is always done to ensure uniqueness of the expressions. This process can be done quite quickly.

```
pseudoFraction :=
2345579801131790854362751370814843424117644570549770038095964332887452245531
1996719229677530820434742068602985136406928886158964000153692762143196723015
5999430033029702007393978659078558205152170198713092853410826825335962454434
0281311926484731789208563757150662371917177308400982024330610776216772820634
4363949283739061026441387072101520115353550440009151571225915883337135032698
8098051364641400474137770449068676809090353794872805217888434388131185224705
5760824638271702281598244411589738109888711109176412397447951841369970410720
09298892351383483898992739497799399689667285315184 33/
1664712420959397341634316090003437490502231774698204427321479299423316001092
2765592072162903350202088054366916349472625185350577714800349724729025353453
3583697681355359834914108345690956852485571468213692585813219890231343118839
9447347002473195024278611609049441002070388437474082345160121203844409382993
3735946972135600445999082477041881856178171753665831882605811450470903668
7159724176466572373412186266194944506096773452713133582603573022094524644920
0170918834827325962809257921639274740872044790047134419764337715663570199239
9524406838281297650744694073655031915519324720737
```

`pseudoFraction` reduces to an integer.

```
pseudoFraction
```

Reducing the fraction built from two 580-digit numbers to an integer one million times takes a few seconds on a year-2005 computer. (We repeat the cancellation very often to obtain a more reliable timing result.)

```
Do[pseudoFraction, {10^6}] // Timing
```

The third important class of numbers is the real numbers with finite accuracy.

`Real`

is the head for floating-point numbers. These are numbers with a decimal point, and they have finite accuracy.

Here is a real number.

```
3.46675890
```

```
Head[%]
```

For a `Real` "zero" (the head is `Real`), we do not get the corresponding simplification `0.0 x` ⟶ `0.0`, which we had above for the `Integer` 0.

```
0.0 arbitraryNumber
```

Given an exact real number—where the word "real" is now interpreted in the usual sense, meaning all of whose unspecified digits are identically 0, it has to be input as an integer. If we want *Mathematica* to treat a number exactly in all future computations, we have to input them as integers or fractions.

We turn now to complex numbers in more detail.

`Complex`

is the head for numbers involving the imaginary unit `I`. Their real and imaginary parts can have the head `Integer`, `Rational`, or `Real`.

If we input a fraction of the form `Complex[...]/Complex[...]`, *Mathematica* will compute its real and imaginary parts and the result will be converted to a number of type `Complex`.

```
(3 + 5 I)/(45 + 67 I)
```

The next complex number has an exact real and an approximate imaginary part.

```
2 + 3. I
```

The real parts of the following fraction are both exact. The collapsed form has an inexact real part.

```
(356 + 78.67 I)/(345 + 89.99 I)
```

Expressions containing numbers as well as symbols or symbolic expressions (like square roots) are not automatically transformed into a normal form.

```
(Sqrt[2] + 78 I)/(3 + Sqrt[3] I)
```

```
(Sqrt[2] - I)/(-Sqrt[2] + I)
```

In the following two inputs the real (imaginary) part is approximative. As a result, the imaginary (real) part autonumericalizes.

```
Sqrt[3] + 2. I
```

```
Sqrt[3.] + 2 I
```

But the following example collapses to one approximative number with the head `Complex`.

```
(Sqrt[2.] - I)/(-Sqrt[7] + 2 I)
```

If a complex number has real and imaginary parts such that one is exact and one has a finite accuracy, the “exactness” of the two constituents remain unchanged.

```
3 + 6.89789 I
```

However, if any computations are performed with such a number, the result will generally involve approximate numbers only.

```
(3 + 6.89789 I)/(4 + 8.9786 I)
```

On the other hand, if we apply an operation that works on the real and imaginary parts separately, the “exactness” of these parts (exact or approximate) will be maintained.

```
3 (3 + 6.89789 I)
```

```
(3 + 6.89789 I) + (2 + 6 I)
```

Sometimes, the real and the imaginary parts *unavoidably* become inexact (see Section 1.5 of the Symbolics [67✱] volume). Here is an example.

```
((* approximate 1 *) 1.0 + 2 I)/(19/3 - 1/6(1 - I Sqrt[3])*
 (1/2 (2963 + 3 I Sqrt[70131]))^(1/3) - (133 (1 + I Sqrt[3]))/
         (3 2^(2/3) (2963 + 3 I Sqrt[70131])^(1/3)))
```

In addition to the elementary (atomic) objects discussed above (numbers with head `Integer`, `Rational`, `Real`, or `Complex`, and symbols with head `Symbol`), one other type of elementary object exists: strings.

`String`

is the head of a string.

Strings can be recognized by their quotes. However, in `OutputForm`, the quotes are not visible.

```
stri = "I am a true string"
```

```
InputForm[stri]
```

```
OutputForm[stri]
```

```
FullForm[stri]
```

`StandardForm`, like `OutputForm` displays no quotes.

```
StandardForm[stri]
```

`TraditionalForm` also does not display the quotes.

```
TraditionalForm[stri]
```

For the current purpose of discussing the most important heads of *Mathematica* expressions, we mainly want to point out the existence of strings; we discuss them and their applications in more detail in Chapter 4. The following constructions involving strings are syntactically correct *Mathematica* expressions but, for most purposes, semantically useless.

```
(6.34 + 34I)["ams"]
```

```
FullForm[%]
```

```
Head[%]
```

```
"acm"[634 + 34.0I]
```

```
FullForm[%]
```

```
Head[%]
```

Approximative numbers can be input in various ways. Here is a short input for machine numbers.

```
5.12 10^-256
```

```
InputForm[%]
```

```
5.12*^-256
```

Here is a number with many digits explicitly written out.

```
2.560000000000000000000000000000000000000000000000000
```

Here, we input this number in a shorter way.

```
2.56`53
```

`InputForm` of this number displays the number of certified digits. (The precision itself is a real number, not an integer; we will discuss this in detail in Chapter 1 of the Numerics volume [66✱].)

```
InputForm[%]
```

Here is a high-precision number with known digits before and after the decimal point.

```
24623000000000000000000000000000000000000000000000\
00000000000000000000000000.000000000000000000
```

Here is the same number input in a shorter way.

```
2.4623`95*^76
```

In general, *number`precision*^base10Exponent* represents a precision digit version of the number $number \times 10^{base10Exponent}$. Here is another example.

```
-123.45`100*^-10
```

precision can be a machine floating-point number (or even a negative number; we discuss this case in Chapter 1 of the Numerics volume [66✱].)

```
-123.45`100.5*^-10
```

```
100.0`-2*^-10
```

We input a number with only four correct digits.

```
8.923`4*^-156
```

```
FullForm[%]
```

For 0, we cannot use this form of inputting because 0 does not have any nontrivial digits. So, the following output will be an exact zero.

```
0.0`4*^-100
```

This is a machine zero.

```
0.000000000000
```

```
InputForm[%]
```

This input also gives a machine zero.

```
0.00000000000000000000000000000000000000000000000000000
```

```
InputForm[%]
```

A number that is known to be zero within $\pm 10^{-n}$ can be input in the form `0``n`. Here is an example shown. In output, such zeros display as `0.` $\times$ `10`$^{-n}$. (Similar to the precision of a number, the accuracy too is internally a real number and not an integer.)

```
0``100
```

```
InputForm[%]
```

```
FullForm[%]
```

High-precision real numbers (meaning numbers having more digits than machine real numbers) are shown in `Input-Form` and `FullForm` in the form *number*`*precision*. *number* is the actual real number, and *precision* is a floating-point approximation of its precision. Because numbers are stored internally in the computer in binary form, a small difference may exist between the input and the internal number for numbers of type `Real` (and `Complex`).

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

■ 2.2.2 Simplest Arithmetic Expressions and Functions

We now examine the elementary arithmetic operations: addition `+`, subtraction `-`, multiplication `*`, division `/`, and exponentiation `^`, as *Mathematica* expressions. We begin with `+`. Here is a simple sum of two summands.

```
3 + x
```

It has the following `FullForm`.

```
FullForm[3 + x]
```

This expression is too small to have an interesting `TreeForm`.

```
TreeForm[3 + x]
```

The next example shows how the order of `x` and `3` in the input differs from the order in the output. Using commutativity, the sum is rearranged into a normalized form. (We discuss the meaning of this in Chapter 4.)

```
OutputForm[x + 3]
```

`StandardForm` will give the same result. In `TraditionalForm`, the two summands get reordered for display. (The internal order does not change.)

```
InputForm[3 + x]
```

```
StandardForm[3 + x]
```

```
TraditionalForm[3 + x]
```

```
FullForm[%]
```

The head of this expression is `Plus`.

```
Head[3 + x]
```

The head of `OutputForm[x + 3]` is also `Plus`, because `OutputForm` acts only as a wrapper for the output.

```
Head[%%]
```

Here is a product of two factors written in three different ways in the input.

```
FullForm[4 y]
```

```
TreeForm[4 y]
```

```
FullForm[4*y]
```

```
OutputForm[y * 4]
```

The multiplication sign appears in the `InputForm` generated by *Mathematica*, but in products input interactively, a space is usually used to improve appearance and readability and to make the input look more like usual mathematical formulas.

```
InputForm[%]
```

```
Head[4 y]
```

The order of the terms is changed for multiplication because it is also commutative. An integer different from 4 does not change the structure `Times[`*integer*`, y]`.

```
FullForm[-4 y]
```

The following sum has three summands.

```
FullForm[3 + x + y]
```

The following product has three factors.

```
FullForm[3 x y]
```

The input `-r` is evaluated to `(-1)*r`. This expression has the head `Times`, which would not happen with `-4` instead of `-r`, because `-4` is *one* number, not a product of `-1` and `4`. `-4` is already parsed as one number.

```
FullForm[-r]
```

Similarly, `1/r` is converted to `r^- 1`.

```
FullForm[1/r]
```

The function `Power` represents all powers.

```
FullForm[r^2]
```

```
OutputForm[r^12]
```

Expressions with a rational exponent lead to a nontrivial tree form. Note the parentheses in the exponents of the input.

```
FullForm[r^(1/2)]
```

```
TreeForm[r^(1/2)]
```

Because of the strong precedence of `Power` over `Times`, we have the product $1/2\,r$ without the parentheses.

```
TreeForm[r^1/2]
```

An alternative way to write `r^(1/2)` is `Sqrt[r]`.

```
InputForm[Sqrt[r]]
```

In output, a square root is usually written as `Sqrt` as opposed to `Power[…, 1/2]`.

```
OutputForm[r^(1/2)]
```

In `StandardForm` and `TraditionalForm`, a square root sign is used.

```
StandardForm[r^(1/2)]
```

```
TraditionalForm[r^(1/2)]
```

The use of `Power` in connection with 0 leads to the following results.

```
0^number
```

```
0.0^number
```

$0^{something}$ stays unevaluated because `number` could be zero (or negative or complex); in which case, the result would be indefinite. When zero is used as the exponent, however, the result is 1 or 1.0 if `number` is nonzero.

```
number^0
```

```
number^0.0
```

`0^0` and `0.0^0.0` are indefinite (or `Indeterminate` in *Mathematica*); we come back to `Indeterminate` in a moment.

```
0^0
```

```
0.0^0.0
```

From the point of view of the *Mathematica* language, $2^{1/2}$ is not a number because its head is not one of the following four: `Integer`, `Real`, `Rational`, or `Complex`. Instead, it is a power, and its head is `Power`.

```
Head[Sqrt[2]]
```

$\sqrt{2}$ could have been thought of as type `AlgebraicNumber`. However, *Mathematica* considers $\sqrt{2}$ to be the result of applying the function `Power` to the integer `2`. The reason is that syntactically we apply the square root function to the argument 2. (*Mathematica* can also handle algebraic numbers; they are `Root`-objects. We will discuss them in Chapter 1 of the Symbolics volume [67✱].) It is not an elementary expression and does not have its own number type. Here is a somewhat more complicated expression with a more complicated `TreeForm`: $3+4\,x-x^3$. The individual summands are

$$\begin{array}{rl} 3 \longrightarrow & \texttt{3} \\ 4\,x \longrightarrow & \texttt{Times[4, x]} \\ -x^3 \longrightarrow & \texttt{Times[-1, Power[x, 3]]} \end{array}$$

```
FullForm[3 + 4 x - x^3]
```

```
OutputForm[3 + 4 x - x^3]
```

```
TreeForm[3 + 4 x - x^3]
```

Here is a summary of the basic arithmetic operations.

`Plus[`*summand*$_1$`, ` *summand*$_2$`, ` …`, ` *summand*$_n$`]`
    or
summand$_1$ + *summand*$_2$ + ⋯ + *summand*$_n$
    gives the sum $summand_1 + summand_2 + \cdots + summand_n$ of the n summands $summand_i$ ($i = 1, \ldots, n$).

`Times[`*factor*$_1$`, ` *factor*$_2$`, ` …`, ` *factor*$_n$`]`
    or
factor$_1$ * *factor*$_2$ * ⋯ * *factor*$_n$
    or
factor$_1$ × *factor*$_2$ × … × *factor*$_n$
    or
factor$_1$ *factor*$_2$ ⋯ *factor*$_n$
    gives the product $factor_1\, factor_2 \cdots factor_n$ of the n factors $factor_i$ ($i = 1, \ldots, n$).

`Power[`*base*`, ` *exponent*`]`
    or
base^*exponent*
    gives the base *base* raised to the exponent *exponent*: $base^{exponent}$.

`Sqrt` is a special case of `Power`.

`Sqrt[`*expression*`]`
    gives the square root of *expression*. `Sqrt[`*expression*`]` is equivalent to *expression*`^(1/2)`.

To the extent that they are defined mathematically, all mathematical functions are implemented for arbitrary complex arguments. Thus, the exponent in `Power` can be a complex number.

```
Power[2.3 + 5.6 I, 2.9 - 8.7 I]
```

Using high-precision numbers, we get a result with more certified digits.

```
Power[2.3`100 + 5.6`100 I, 2.9`100 - 8.7`100 I]
```

For a symbolic base z, the following product of three powers collapses into one power.

```
z^(1/2) z^(1/3) z^(1/4)
```

Next, we plot the real and imaginary parts and the absolute value of $(-2)^x$ for $-3 < x < 5$ [58✱].

```
Needs["Graphics`Legend`"]
```

```
Plot[(* the curves *)
     {Re[(-2)^x], Im[(-2)^x], Abs[(-2)^x]}, {x, -3, 5},
     PlotStyle -> {{AbsoluteThickness[0.5], AbsoluteDashing[{4, 4}]},
                   {AbsoluteThickness[0.5], AbsoluteDashing[{2, 2}]},
                   {AbsoluteThickness[0.5]}}, Axes -> None,
    (* the legend *)
    PlotLegend -> (StyleForm[#, FontFamily -> "Courier",
                    FontWeight -> "Plain", FontSize -> 10]& /@
                   {" Re[(-2)^x]", " Im[(-2)^x]", "Abs[(-2)^x]"}),
    (* further options *)
    LegendPosition -> {-0.5, -0.3}, LegendSize -> {0.92, 0.29},
    PlotRange -> All, Frame -> True, FrameLabel -> {"x", None}]
```

Here, we observe the behavior of `Plus` and `Times` when only one argument exists, or none at all.

```
Plus[plus]

Plus[]

Times[times]

Times[]
```

(For the moment, we just want to take note of this behavior; Chapter 3 explains why `Plus` and `Times` behave this way.)

We now discuss the head(s) of user-defined symbols and built-in functions. As noted previously, a user-defined symbol `x` has the head `Symbol`.

```
Head[x]
```

`Symbol`
    is the head for a symbol.

The system functions discussed above also have this head.

```
Head[Plus]

Head[TreeForm]
```

Note that *Mathematica* also understands the following expressions but immediately rewrites them.

`Subtract[`*a*, *b*`]` means *a* − *b*
    and becomes `Plus[`*a*`, Times[-1, `*b*`]]`.

`Divide[`*c*`, `*d*`]` means *c*/*d*
    and becomes `Times[`*c*`, Power[`*d*`, -1]]`.

`Minus[`*expression*`]` means −*expression*
    and becomes `Times[-1, `*expression*`]`.

Here are three simple examples.

```
Subtract[α, β]

Divide[α, β]

Minus[α]
```

If not explicitly entered into *Mathematica*, *Mathematica* will never generate expressions with head `Subtract`, `Divide`, and `Minus`.

Because the forms $a-b$, a/b, and $-a$ are immediately rewritten (through evaluation) and stored in the rewritten form, we cannot get them back using `FullForm`. (We discuss a way around this in Chapter 3.)

```
FullForm[a - b]

FullForm[α/β]

FullForm[-α]
```

An analogous assertion also holds for `InputForm`, `TreeForm`, `OutputForm`, and almost every other built-in and user-defined function. If we input some "uncomputed" expression, the result of these formats does not return the input expression, but rather the format of the result computed by *Mathematica*. This strategy of stepwise computation from the inside out holds for every expression in *Mathematica*. We come back to this in detail in Chapter 4. So, the result of the following is just 0 and not `1 - (-(-1))` and `Plus[1, Times[-1, Times[-1, -1]]]`.

```
InputForm[1 - (-(-1))]

FullForm[1 - (-(-1))]
```

In Chapter 3, we discuss how to get the `InputForm` of such expressions and the functions that are exceptions to this rule.

Be aware that in `TraditionalForm` inputs, the *Mathematica* precedences and groupings for operators still hold. So $\varepsilon/4\,\pi$ is interpreted as `Times[1/4,π,ε]`. One has to add explicit parentheses in $\varepsilon/(4\,\pi)$ to get `Times[1/4,`π^{-1}`,ε]`.

Note which expressions are simplified (or converted) and how they are simplified in the following examples. We will discuss some of these examples in more detail shortly.

```
Sqrt[9/25]

Sqrt[2] + Sqrt[3]

(11^7)^(2/7)

(9999^888)^(1/444)

I^(1. I)

(8/27)^(1/3)

(Sqrt[12] - Sqrt[20])^2/4

(1 + Sqrt[2])^2

(2 + (-121)^(1/2))^(1/3)

(Sqrt[2] + Sqrt[7])^2

(Sqrt[2] + Sqrt[8])^2

Sqrt[18] (8.0)^(1/3)

Sqrt[z^2]

Sqrt[1 + x]/(1 + x)

(a^(1/3))^(1/2)

2 2^w

2^w1 2^w2
```

```
(-2) (-a - b)

(-1) (-a - b)

8.0^(1/3)

8.0^(1.0/3.0)

(1 + 0.0)^(0 - 0.0)

(0.0 I)^(0.0 + 0.0 I + 1)

2 + ((Sqrt[2] + I)^2 - 2 - 2 Sqrt[2] I + 1) I + 0.0

2 + ((Sqrt[2] + I)^2 - 2 - 2 Sqrt[2] I + 1) I + 0.0 I
```

Σ (* session summary *) **TMGBs`PrintSessionSummary[]**

■ 2.2.3 Elementary Transcendental Functions

The elementary transcendental functions are e^x, $\ln x$, and trigonometric and hyperbolic functions. (We will discuss the inverses of the trigonometric and hyperbolic functions in Subsection 2.2.5.) (For a more mathematical definition of the term “elementary function”, see [71✱].) Using *Mathematica*’s naming conventions, the exponential function is written `Exp[x]`.

```
Exp[1.89]
```

Because all functions in *Mathematica* also work with complex arguments, we can evaluate the exponential function for a complex argument.

```
Exp[1.89 + 9.87 I]
```

We can also plot the exponential function. The next plot shows values along the real axis. (We discuss `Plot` and the related graphics functions `Plot3D` and `ContourPlot` in detail in the Graphics volume [65✱].)

```
Plot[Exp[x], {x, -1, 2}, AxesLabel -> {"x", "Exp[x]"},
     PlotStyle -> Thickness[0.01]]
```

The function $e^{1/z}$ is much more interesting than is e^z, especially if we look at the real part of the function in a region of the complex plane near the origin.

```
Plot3D[Re[Exp[1/(x + I y)]], {x, -2.001, 2}, {y, -2.001, 2},
       PlotPoints -> 60]
```

Magnifying the plot of $e^{1/z}$ in the neighborhood of $z = 0$ is especially interesting because of the essential singularity at $z = 0$. The height of the plotted points (the function value) is proportional to the real part, and the color is related to the phase; we show only function values in the range $-1 < \mathrm{Re}(e^{1/z}) < 8$. (To avoid the generation of error messages caused by too large numbers to be displayed, we turn off the corresponding message with `Off[Plot3D::gval]`.)

```
Off[Plot3D::gval];

Plot3D[{Re[Exp[1/(x + I y)]], Hue[Arg[Exp[1/(x + I y)]]]},
       {x, -0.02, 0.022}, {y, -0.04, 0.042},
       PlotRange -> {-1, 8}, PlotPoints -> 120, Mesh -> False]

Off[Plot3D::gval];
```

Next, we show the lines where the imaginary part is constant. (Here we use a random coloring; the details of this plot will be discussed in Chapter 3 of the Graphics volume [65✱].)

```
Module[{cp, cls, L = 0.02},
          (* an initial contour plot *)
          cp = ContourPlot[Im[Exp[1/(x + I y)]], {x, -L, L}, {y, -L, L},
                           PlotPoints -> 400, DisplayFunction -> Identity] /.
          (* replace large high-precision numbers by biiig machine numbers *)
             z_?(Abs[#] > $MaxMachineNumber&) :> Sign[z] $MaxMachineNumber/2;
          (* homogeneously distributed contour lines *)
          cls = #[[100]]& /@ Partition[Sort[Flatten[cp[[1]]]], 800];
          (* the final contour plot *)
          ListContourPlot[cp[[1]], MeshRange -> {{-L, L}, {-L, L}},
                          Contours -> cls, ContourLines -> False,
                          ColorFunction -> (Hue[Random[]]&),
                          AspectRatio -> Automatic, FrameTicks -> None]]
```

The reason for this wild behavior of $e^{1/z}$ near $z = 0$ is explained by the Theorem of Picard.

Mathematical Remark: Theorem of Picard

If $f(z)$ is a one-to-one analytic function in the neighborhood of a point $z = a$, and if it has an essential singularity there, $f(z)$ takes on every arbitrary finite value, with at most one exception, in every neighborhood of a. See any textbook on function theory, for example, [57✱], [13✱], [36✱], and [47✱].

We can use not only the exponential function in the complex plane, but also all mathematical functions.

As long as they make sense (meaning an analytic continuation is possible), all functions in *Mathematica* are available for arbitrary complex numbers.

Here is an important remark concerning the arguments of inverse trigonometric functions.

The arguments of trigonometric functions are always given in radians. To deal with arguments in degrees, see the next subsection.

This fact means we have the following results.

```
Sin[3.1415926535897932385]

Sin[3.1415926535897932385/3]
```

Mathematica includes the following elementary transcendental functions. (The inverse trigonometric and hyperbolic functions will be discussed in Subsection 2.2.5. Here we keep the exp-log pair together [12✱], [44✱].)

`Exp[`*expression*`]`
    gives the exponential function $e^{expression}$.

`Log[`*expression*`]`
    gives the natural logarithm ln(*expression*).

`Log[`*base*`, `*expression*`]`
    gives the logarithm of *expression* to the base *base*.

`Sin[`*expression*`]`

gives the sine function sin(*expression*).

`Cos[`*expression*`]`

gives the cosine function cos(*expression*).

`Tan[`*expression*`]`

gives the tangent function tan(*expression*).

`Cot[`*expression*`]`

gives the cotangent function cot(*expression*).

`Sec[`*expression*`]`

gives the secant function sec(*expression*). ($\sec(z) = 1 / \cos(z)$)

`Csc[`*expression*`]`

gives the cosecant function csc(*expression*). ($\csc(z) = 1 / \sin(z)$)

`Sinh[`*expression*`]`

gives the hyperbolic sine function sinh(*expression*).

`Cosh[`*expression*`]`

gives the hyperbolic cosine function cosh(*expression*).

`Tanh[`*expression*`]`

gives the hyperbolic tangent function tanh(*expression*).

`Coth[`*expression*`]`

gives the hyperbolic cotangent function coth(*expression*).

`Sech[`*expression*`]`

gives the hyperbolic secant function sech(*expression*). ($\operatorname{sech}(z) = 1 / \cosh(z)$)

`Csch[`*expression*`]`

gives the hyperbolic cosecant function csch(*expression*). ($\operatorname{csch}(z) = 1 / \sinh(z)$)

We stop to take a quick look at the somewhat less frequently used functions sec, csc, sech, and csch. Creating the following plots (axes, labels, width of lines, removal of vertical lines, etc.) is discussed in detail in Chapter 1 of the Graphics volume [65✱].

```
Module[{ε  = 10^-10, i},
Show[GraphicsArray[
Block[{$DisplayFunction = Identity},
(* left picture *)
Show[Table[(* for avoiding vertical lines *)
      Plot[#[[1]][x], {x, i Pi/2 + ε, (i + 1) Pi/2 - ε},
           PlotStyle -> Thickness[0.01]], {i, -4, 3}],
     (* setting options so that plot looks nice *)
      PlotRange -> {All, {-10, 10}},
      TextStyle -> {FontFamily -> "Times", FontSize -> 9},
      Ticks -> {{#[[1]], StyleForm[#[[2]], FontSize -> 9]}& /@
       {{-2Pi, "-2π"}, {-Pi, "-π"}, {0, "0"}, {Pi, "π"}, {2Pi, "2π"}},
       Automatic}, AxesLabel -> {StyleForm[TraditionalForm[x]], None},
      PlotLabel -> StyleForm[TraditionalForm[#[[1]][x]],
                             FontWeight -> "Bold",
       FontSize -> 11]]& /@ {{Sec, "sec"}, {Csc, "csc"}}]]];
(* right picture *)
Show[GraphicsArray[
Block[{$DisplayFunction = Identity},
Show[{Plot[#[[1]][x], {x, -4, -ε}, PlotStyle -> Thickness[0.01]],
      Plot[#[[1]][x], {x,  ε,  4}, PlotStyle -> Thickness[0.01]]},
      (* setting options so that plot looks nice *)
      DisplayFunction -> Identity, PlotRange -> {All, #[[3]]},
      TextStyle -> {FontFamily -> "Times", FontSize -> 9},
      AxesLabel -> {StyleForm[TraditionalForm[x]], None},
      PlotLabel -> (* function label *)
       StyleForm[TraditionalForm[#[[1]][x]], FontWeight -> "Bold",
      FontSize -> 11]]& /@
          {{Sech, "sech", {0, 1}}, {Csch, "csch", {-10, 10}}}]]]]
```

We show now an interesting graphic based on $x \to \sec(x+\alpha)$ iterations. Here α is a parameter. We will iterate the function 2000 times and discard the first 200 iterations. The resulting functions are in general wildly oscillating as a function of α, but for certain α only a small number of different numerical values occur for the iterates. The following graphic shows the parameter interval $1.026 \le \alpha \le 1.040$. We see many of the well-known bifurcations often shown for the quadratic map.

```
With[{ppi = 500, pp = 2000},
Show[Graphics[{PointSize[0.002], Table[Point[{α, #}]& /@
      Drop[NestList[N[Sec[# + α]]&, -1/2., ppi], 200],
      (* small α-interval *) {α, 1.026, 1.040, 0.014/pp}]}],
      Frame -> True, PlotRange -> {-2.25, -0.99}]]
```

When exact arguments lead to exact values for these transcendental functions, *Mathematica* gives an exact result. The same rule holds for arguments of type `Integer`, `Rational`, and `Complex`, as well as for algebraic and transcendental arguments. If the value is exact and *Mathematica* knows no special value, the input remains unchanged—this result is still an exact representation of the expression.

```
Exp[I Pi 2]

Log[1]

Log[8, 2]

Sin[2]
```

Numerical values (actually numerical approximations) of an analytic expression (an “exact” number) can be obtained using `N`. The function `N` (as well as any other one-argument function) can be applied in three different ways.

`N[`*expression*`]`
    or
expression `// N`
    or
`N @` *expression*
    computes the numerical value of *expression*.

Here, we use all three approaches to compute a numerical value of sin(2).

```
N[Sin[2]]
```

```
Sin[2] // N
```

```
N @ Sin[2]
```

If a function is called with numerical variables, it will then, in general, “automatically” produce a numerical value.

```
Sin[2.0]
```

```
Sin[N[2]]
```

```
Sin[N @ 2]
```

```
Sin[2 // N]
```

Note that these three ways of applying a function to an argument can be used for any function, built-in or user-defined, explicitly computable or not explicitly computable. Here is an example using a user-defined symbol a𝛼𝕒𝔞.

```
a𝛼𝕒𝔞 @ argument
```

```
argument // a𝛼𝕒𝔞
```

```
a𝛼𝕒𝔞[argument]
```

If we apply `N` to `0`, we get machine number `0.` (for brevity *Mathematica* uses `0.` instead of `0.0` in output form).

```
N[0]
```

```
Head[%]
```

There is no high-precision 0 with a finite number of correct digits. (We discuss the reason in detail in Chapter 1 of the Numerics volume [66✱].)

```
N[0, 100]
```

```
Head[%]
```

Only “to which size” a number is zero can be indicated.

```
0``100
```

The head of an approximative `0.0` is `Real`, and the head of `0.0 + 0.0 I` is `Complex`.

```
Head[0.0]
```

```
Head[0.0 + 0.0 I]
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

■ 2.2.4 Mathematical Constants

The mathematical constants e, π, γ, and so [26✱] on are exact numbers in the mathematical sense. However, from a programming standpoint, these constants are symbols (with head `Symbol`) in *Mathematica*. This is, in a certain sense, analogous to the treatment of the algebraic numbers $\left(2^{1/2}, 5^{1/3} - 7^{1/4}, \ldots\right)$ discussed above.

The fundamental rule for calculation with complex numbers is $i^2 = -1$. (For historical and mathematical details about i, see [46✱].)

```
I^2
```

`I` is a number with head `Complex`.

```
Head[I]
```

```
FullForm[I]
```

The square root of -1 is i. (Note that in the following, the only root given is $+i$.)

```
(-1)^(1/2)
```

`I`

represents the imaginary unit i, that is, $i^2 = -1$.

Next, we will discuss π.

For certain simple rational fractions of π = `Pi` (more exactly, for integer multiples of $\pi/4$ and $\pi/6$) the trigonometric functions `Sin`, `Cos`, `Tan`, `Cot`, `Sec`, and `Csc` give exact values.

```
Sin[Pi]
```

```
Sin[Pi/6]
```

```
Tan[45 Pi/4]
```

```
Head[Pi]
```

If the input contains an inexact number, the output will “collapse” to an inexact number whenever possible.

```
Sin[1.5 Pi]
```

`Pi`

represents the exact irrational number π.

For many details on the history, different representations, etc. of π, see [3✱], [9✱].

Here is a fraction that equals π to about 50 digits.

```
N[Pi - 23294267674065827396789607/7414795692066647773964845, 60]
```

Now, we can look at the values of the trigonometric functions for special fractions of π in more detail. (We discuss how to produce this kind of table in Chapter 6. $\tilde{\infty}$ stands for `ComplexInfinity`, to be discussed shortly.)

```
With[{functions = {Sin, Cos, Tan, Cot, Sec, Csc},
      args = {Pi/2, Pi/3, Pi/4, Pi/5, Pi/6, Pi/10, Pi/12}},
     TableForm[(* this forms all combinations of
        functions and arguments *)
          Outer[#1[#2]&, functions, args] /.
          ComplexInfinity -> OverTilde[DirectedInfinity[1]],
          TableHeadings -> {functions, args}, TableSpacing -> 0.45,
          TableAlignments -> {Center, Center}]]
```

The last evaluation of *trigonometricFunction* `[Pi/`*integer*`]` happens automatically for *integer*=1, 2, 3, 4, 5, 6, 10, and 12. Using the function `FunctionExpand` (to be discussed in Chapter 3 of the Symbolics volume [67✱]), more expressions of the form *trigonometricFunction* `[Pi/`*integer*`]` can be expressed in nested radicals.

```
Sin[Pi/9] // FunctionExpand
```

```
Sin[Pi/256] // FunctionExpand
```

```
Cos[Pi/17] // FunctionExpand
```

A close relative of π is `Degree`.

`Degree`

stands for one degree (1/360 of a full circle).

With `Degree`, we can input the argument of the trigonometric functions in degrees. `Degree` has precisely the value $2\pi/360$. The use of `N` results in a numericalized version of the expression.

```
Degree // N
```

```
2 Pi/360 // N
```

The expression `30 Degree` is `Times[30,Degree]` in `FullForm`.

```
Sin[30 Degree] // N
```

Mathematica does, of course, not differentiate between arguments of trigonometric functions *someInteger* `Degree` or `Pi/(180/`*someInteger*`)`.

```
Tan[30 Degree]
```

```
Tan[Pi/6]
```

When possible, trigonometric functions of arguments containing general variables will be simplified. Here are a few examples—observe the results only, not the programming.

```
TableForm[Outer[#1[#2]&, {Cos, Sin, Tan, Cot, Sec, Csc},
                         {Pi/2 + x, Pi + x, 3/2 Pi + x}],
          (* table headers *)
          TableHeadings -> {{Cos, Sin, Tan, Cot, Sec, Csc},
                   {"Pi/2 + x \n", "Pi + x\n", "3/2 Pi + x\n"}},
          TableAlignments -> {Right, Center}]
```

Trigonometric functions that can be expanded into sums of several terms are not automatically converted. (Of course, *Mathematica* supplies functions to carry out such expansions, as discussed in Chapter 3.)

```
Cos[Pi/3 + x]
```

```
Sin[Pi/4 - x]
```

The famous Euler identity connects the numbers e [40✱], i, and π.

```
Exp[I Pi]
```

The following "related" construction $\pi^{i\,e}$ gives "almost" -1.

```
Pi^(E I) // N
```

The number e itself is denoted in *Mathematica* by `E`.

```
Exp[1]
```

```
Log[E]
```

`E`

represents the exact irrational number e.

It is well known that e can be defined as the limit value $\lim_{n\to\infty}(1 + 1/n)^n$.

We can examine a plot of the convergence of this sequence by looking at the base 10 logarithm of the difference (for bounds on the difference, see [49✱]).

```
Plot[Log[10, E - (1 + 1/n)^n], {n, 1, 1000},
     AxesLabel -> {n, Log[E - (1 + 1/n)^n]}]
```

We now consider another mathematical constant.

```
2 N[Cos[Pi/5]]
```

It is called the golden ratio or, in the *Mathematica* naming convention, `GoldenRatio`.

```
GoldenRatio // N
```

The on-line explanation is given below.

```
?GoldenRatio
```

`GoldenRatio`

gives the exact golden ratio.

Mathematical Remark: Golden Ratio

The golden ratio ϕ arises by dividing a segment of length a into two parts so that the ratio of the length of the larger part x to the full length a is equal to the ratio of the length of the smaller part $a - x$ to the length of the larger part x, which means $x/a = (a-x)/x$.

Solving for a/x gives $\phi = a/x = \left(1+\sqrt{5}\right)/2$. For a detailed discussion of the golden ratio, with many interesting graphics applications, see [70✱], [14✱], and [68✱]. For misconceptions about the history and use of golden ratio, see [41✱], [42✱], and [50✱].

The equality of the two numbers `2 Cos[Pi/5]` and `GoldenRatio` is no accident: Many function values of sin, cos, tan, and cot corresponding to small fractions ($1/5$, $1/10$, $1/12$, …) of π can be represented in terms of the golden ratio (e.g., $\sin(\pi/10) = \left(5^{1/2}-1\right)/4 = (\phi-1)/2$).

```
Sin[Pi/10]
```

Another important mathematical constant is γ.

```
?EulerGamma
```

```
EulerGamma
```

represents the exact Euler number γ.

Mathematical Remark: γ

The Euler number γ is defined as the following limit value [30✱]:

$$\lim_{n\to\infty}\left(\frac{1}{1}+\frac{1}{2}+\frac{1}{3}+\cdots+\frac{1}{n}-\ln(n)\right)$$

This can also be expressed as the following sum:

$$\gamma=\sum_{k=1}^{\infty}\left(\frac{1}{k}-\ln\left(1+\frac{1}{k}\right)\right)$$

The Euler number γ arises frequently in computing definite integrals.

Here are the first few partial sums of this sequence for computing γ. It converges extremely slowly. (For a simple method to accelerate the convergence of this series, see [63✱].)

```
Do[Print[NSum[1/i, {i, 1, n}] - N[Log[n]]], {n, 1, 12}]
```

The following graphic shows $\log_{10}\left(\left|\gamma-\left(\ln(n)-\sum_{k=1}^{n}1/k\right)\right|\right)$. For $n=10^4$, the direct summation gives about four correct digits for γ.

```
ListPlot[Log[10, Abs[EulerGamma - MapIndexed[#1 - Log[#2[[1]]]&,
                         Rest[FoldList[Plus, 0, 1./Range[10^4]]]]]],
         Frame -> True, Axes -> False]
```

Here is a much more efficient series representation for γ [34✱].

$$\gamma = 1-\lim_{n\to\infty}\sum_{k=1}^{12n+1}\frac{(-1)^{k-1}\,n^{k+1}}{(k-1)!\,(k+1)}\left(\log(n)-\frac{1}{k+1}\right)$$

By summing less than 1000 terms, we get about 34 digits if γ.

```
γK[n_] := 1 - Sum[(-1)^(k - 1) n^(k + 1)/((k - 1)! (k + 1))
                   (Log[n] - 1/(k + 1)), {k, 12n + 1}]

γK[N[83, 80]]

% - EulerGamma // N
```

And here is a simple program that allows to calculate thousands of digits of γ. It is based on the approximate summation of alternating series $\sum_{k=0}^{\infty}(-1)^k\,a_k$ [17✱]. We make use of the series $\sum_{k=1}^{\infty}(-1)^k\log(k)/k = \gamma\log(2)-\log^2(2)/2$.

```
(* sum n terms of the alternating series
   Sum[(-1)^k summand[k], {k, 0, Infinity}] *)
sumAlternatingSeries[summand_, n_] :=
Module[{d = (3 + Sqrt[8])^n, b = -1, c = -d, s = 0},
        d = (d + 1/d)/2; b = -1; c = -d; s = 0;
      Do[c = b - c; s = Together //@ (s + c summand[k]);
         b = (k + n)(k - n) b/((k + 1/2) (k + 1)), {k, 0, n - 1}];
      s/d]

(* approximate EulerGamma using n series terms *)
γSumApproximation[n_, prec_] :=
Block[{$MaxExtraPrecision = prec, c2},
      c2 = sumAlternatingSeries[(* the shifted series terms *)
                          Function[k, N[-Log[k + 1]/(k + 1), prec]], n];
      c2/Log[2] + Log[2]/2]
```

Using 1300 terms gives us more than 1000 digits of γ in seconds.

```
γSumApproximation[1300, 1010] - EulerGamma // Timing
```

By avoiding numericalization, we can even obtain symbolic approximations of γ.

```
γSumApproximation[22, Infinity] // Simplify
```

Twenty series terms give about 19 correct digits.

```
N[% - EulerGamma, 20]
```

In *Mathematica*, ∞ is written as `Infinity`, and it can be considered to be a mathematical constant in a certain sense. It can also be given as an argument of functions.

```
Exp[Infinity]
```

```
Exp[-Infinity]
```

`Infinity` has an “interesting” internal form.

```
FullForm[Infinity]
```

`-Infinity` evaluates to the corresponding expression `DirectedInfinity[-1]`.

```
FullForm[-Infinity]
```

In outputs, the last expression formats as $-\infty$.

```
%
```

Infinity in *Mathematica* comes in various “flavors”.

`DirectedInfinity[`*z*`]`

represents a numerically infinite quantity in the direction of the complex number *z*.

`DirectedInfinity[]`

or

`ComplexInfinity`

represents a numerically infinite quantity in an unknown direction in the complex plane.

The value of 1 / *someFlavorOfInfinity* is 0, independent of the direction in the complex plane.

```
1/ComplexInfinity
```

A variety of mathematical operations can be performed with `Infinity` [7✱]. For example, it can appear as the limit in a summation or as the argument in various special functions. We will encounter such cases quite often throughout the *GuideBooks*.

```
DirectedInfinity[1 + I] DirectedInfinity[I]
```

For `DirectedInfinity` a difference exists between the `OutputForm` and the `FullForm`.

```
DirectedInfinity[I]
```

```
FullForm[%]
```

```
OutputForm[%]
```

The following expression is so "badly undetermined" that it even generates an error message.

```
1/0
```

If this $1/0$ occurs as the limit value of $\lim_{t\to t_0} f(t)/g(t)$, it might be possible to determine the direction of the resulting infinity in the complex plane. `DirectedInfinity[1]` is a "positive real infinite number".

`Infinity`
or
`DirectedInfinity[1]`
represents a numerically infinite quantity in the direction of the positive real axis in the complex plane.

`Infinity` possesses no numerical value of its own.

```
N[Infinity]
```

Often, a calculation does not lead to a unique result. For example, in computing $e^\infty - e^{\infty^2}$ in *Mathematica* via `Exp[Infinity] - Exp[Infinity^2]`, first, the two expressions `Exp[Infinity]` and `Exp[Infinity]^2` are formed, and then the two resulting values of `Infinity` are subtracted. `Infinity - Infinity` is not uniquely defined, because they could be of very "different sizes", which at this point, *Mathematica* has already "forgotten". Here is an illustration.

```
Exp[Infinity] - Exp[Infinity^2]
```

The following input gives the same result, of course.

```
Exp[Infinity] - Exp[Infinity]
```

```
Infinity - Infinity
```

The use of the function `Limit`, discussed in Chapter 1 of the Symbolics volume [67✱], often allows the handling of such expressions in a more sensible way.

`Indeterminate`
represents a numerically indefinite quantity.

We should note the following in dealing with quantities that can be infinite. On the one hand, we have the following obvious result.

```
Infinity - Infinity
```

On the other hand, to every symbolic quantity, an arbitrary value, including `Infinity`, can be assigned.

```
arbitraryQuantity - arbitraryQuantity
```

Of course, we cannot do without $x - x = 0$, because then hardly any expressions could be simplified. The analogous situation holds for functions of `Infinity`, in contrast to the example above.

```
arbitraryFunction[Infinity] - arbitraryFunction[Infinity]
```

For many functions *f*, *f*`[Infinity]` will not be `Infinity` or `Indeterminate` and the last example makes sense. *Mathematica* will always assume that a variable or a function value is a "generic" complex number; this means it is not 0 or not a flavor of infinity. (If we want the property `f[Indeterminate] = Indeterminate`, then we could give the function `f` the attribute `NumericFunction`. This will be discussed in Chapter 3.)

```
SetAttributes[f, NumericFunction];
f[Indeterminate]
```

The product of nearly every *Mathematica* expression and 0 is 0. Exceptions to this rule are flavors of `DirectedInfinity` and `Indeterminate`.

```
(* use lower case infinity *)
0 infinity

0 DirectedInfinity[2]

0 Indeterminate
```

And 0^0 cannot be uniquely defined either [35✱].

```
0^0
```

The following expressions all evaluate to `Indeterminate`.

```
Infinity - Indeterminate

Indeterminate - Indeterminate

0^Indeterminate

Indeterminate^0
```

But be aware that `DirectedInfinity` and `Indeterminate` have to occur explicitly inside such products. The product of 0 and a "hidden infinity" returns 0.

```
0 (* a hidden infinity of the form 1/0 *)*
((Pi - 1)^2 - (Pi^2 - 2Pi + 1))^-1
```

More mathematical constants are available in *Mathematica*, but for our purposes, we end here.

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

■ 2.2.5 Inverse Trigonometric and Hyperbolic Functions

We now look at the inverse functions corresponding to the trigonometric and hyperbolic functions. Whenever possible, we get exact results.

```
ArcSin[0]

ArcTan[1]
```

With a real number as the argument given with a decimal point, the result also has a decimal point (the result can be real or complex).

```
ArcSin[0.78]
```

When |*argument*| > 1, the result for arcsin and arccos is complex.

```
ArcSin[5.78]
```

If the input contains more certified digits such that *Mathematica*'s own high-precision arithmetic is used, a more precise result will be returned.

```
ArcSin[5.780000000000000000000000000000000000000000000000]
```

Because the trigonometric functions are multivalued, in general, arcsin(sin(x)) ≠ x. Here is an example for such a function.

```
ArcSin[Sin[5.78]]
```

The inverse functions for the trigonometric and hyperbolic functions only produce values on the principal branch.

`ArcSin[`*expression*`]`

gives the arcsine function arcsin(*expression*). For real arguments satisfying |*expression*| > 1, the result lies in the interval $[-\pi/2, \pi/2]$.

`ArcCos[`*expression*`]`

gives the arccosine function arccos(*expression*). For real arguments satisfying |*expression*| > 1, the result lies in the interval $[0, \pi]$.

`ArcTan[`*expression*`]`

gives the arctangent function arctan(*expression*). For real arguments *expression*, the result lies in the interval $[-\pi/2, \pi/2]$. (The endpoints are attained when the argument is $\pm\infty$.)

```
ArcTan[Infinity]
```

```
ArcTan[-Infinity]
```

`ArcTan` can also be called with two variables.

`ArcTan[`$coordinate_x$`, `$coordinate_y$`]`

gives the polar angle of a point P in the x,y-plane with the coordinates $P = \{coordinate_x, coordinate_y\}$. The result lies in the interval $(-\pi, \pi]$ for real $coordinate_x$, $coordinate_y$. (The right endpoint corresponds to points on the negative real axis.)

We now look at the coordinates of a point, which moves counterclockwise around the origin in steps of 45 degrees. (The == represents mathematical equality; we will discuss it in detail in Chapter 5.)

```
Print[#, " == ", ToExpression[#]]& /@
{"ArcTan[ 1,  0]", "ArcTan[ 1/2,  1/2]",
 "ArcTan[ 0,  1]", "ArcTan[-1/2,  1/2]",
 "ArcTan[-1,  0]", "ArcTan[-1/2, -1/2]",
 "ArcTan[ 0, -1]", "ArcTan[ 1/2, -1/2]"};
```

Mathematica supports three more inverse trigonometric functions: `ArcCot`, `ArcSec`, and `ArcCsc`.

`ArcCot[`*expression*`]`

gives the arccotangent function arccot(*expression*). For a real argument, the result lies in the interval $[-\pi/2, \pi/2]$. (The endpoints are attained for $\pm\infty$ as the argument.)

`ArcSec[`*expression*`]`

gives the arcsecant function arcsec(*expression*). For a real argument, the result lies in the interval $[0, \pi]$. (The endpoints are obtained for ± 1 as the argument.)

`ArcCsc[`*expression*`]`

gives the arccosecant function arccos(*expression*). For a real argument, the result lies in the interval $[-\pi/2, \pi/2]$. (The endpoints are obtained for ± 1 as the argument.)

Trigonometric functions of inverse trigonometric functions are simplified. Here are all possible combinations. (We are only interested in the output, and not the input here. For space reasons, we use `InputForm` as the format type of the output.)

```
Outer[(* forming all combinations of trig[inverseTrig] *)
      (ToString[#1] <> "[" <> ToString[#2] <> "[z]] == " <>
       ToString[InputForm[#1[#2[z]]]])&,
         {Sin, Cos, Tan, Cot, Sec, Csc},
         {ArcSin, ArcCos, ArcTan, ArcCot, ArcSec, ArcCsc}] //
                                           Flatten // TableForm
```

Note that because of the multivaluedness of the inverse trigonometric functions expressions of the form *inverseTrigono*⋱ *metricFunction*`[`*trigonometricFunction*`[`*z*`]]`, do not “simplify” to *z*.

```
ArcSin[Sin[z]]
```

```
ArcSin[Cos[z]]
```

We also have inverse functions for the hyperbolic functions.

```
ArcSinh[2.718]
```

For purely imaginary arguments, the hyperbolic functions reduce to known trigonometric functions.

```
ArcTanh[I]
```

The hyperbolic function sinh is also multivalued (with a purely imaginary period).

```
Sinh[3 + 2 Pi I]
```

```
Sinh[3 + 24 Pi I]
```

Here is a list of all inverse hyperbolic functions.

`ArcSinh[`*expression*`]`

gives the inverse hyperbolic sine function arcsinh(*expression*).

`ArcCosh[`*expression*`]`

gives the inverse hyperbolic cosine function arccosh(*expression*).

`ArcTanh[`*expression*`]`

gives the inverse hyperbolic tangent function arctanh(*expression*).

`ArcCoth[`*expression*`]`

gives the inverse hyperbolic cotangent function arccoth(*expression*).

`ArcSech[`*expression*`]`

gives the inverse hyperbolic secant function arcsech(*expression*).

`ArcCsch[`*expression*`]`

gives the inverse hyperbolic cosecant function arccsch(*expression*).

(For inverses of elementary functions in general, see [60✱].)

Here are the results of applying a hyperbolic function to an inverse hyperbolic function (similar to the trigonometric case above).

```
Outer[(* forming all combinations of hyp[inverseHyp] *)
      (ToString[#1] <> "[" <> ToString[#2] <> "[z]] = " <>
       ToString[InputForm[#1[#2[z]]]])&,
          {Sinh, Cosh, Tanh, Coth, Sech, Csch},
          {ArcSinh, ArcCosh, ArcTanh, ArcCoth,
           ArcSech, ArcCsch}] // Flatten // TableForm
```

If we look at an inverse function in the complex plane, that is, with a complex argument, its absolute value is not a smooth function. Because complex-valued functions of complex variables are hard to plot in two or three dimensions, we first consider five other important operations on complex numbers: determining the real part, the imaginary part, the absolute value, the phase, and the conjugation.

`Re[`*complexNumber*`]`

gives the real part of the complex number *complexNumber* (with head `Complex`).

`Im[`*complexNumber*`]`

gives the imaginary part of the complex number *complexNumber* (with head `Complex`).

`Abs[`*complexNumber*`]`

gives the absolute value of the complex number *complexNumber* (with head `Complex`).

`Arg[`*complexNumber*`]`

gives the argument (phase angle) of the complex number *complexNumber* (with head `Com`⋱
`plex`). The result lies in $(-\pi, \pi]$ (the value $-\pi$ is never returned, real negative *complexNum*⋱
ber give π). If *Mathematica* cannot find the value of `Arg[`*complexNumber*`]` for complex numbers with rational (or exact symbolic irrational) real and imaginary parts, the result appears in the form `ArcTan[`*realPart*`,` *imaginaryPart*`]`.

`Conjugate[`*complexNumber*`]`

gives the complex conjugate $a - i\,b$ of *complexNumber* $= a + i$ (a, $b \in \mathbb{R}$).

Here are a few simple examples.

```
z = 3.98 + 789 I
```

```
Re[z]
```

```
Im[z]
```

```
Abs[z]
```

```
Arg[z]

Conjugate[z]
```

With exact values for the argument, these functions produce exact results whenever possible.

```
z = 3456/7891 + 7876/653 I

Re[z]

Im[z]

Abs[z]

Arg[z]
```

As soon as an approximate element is present, we get an approximate result.

```
Arg[23/45 + 7.89 I]
```

Note, however, that if either the real or the imaginary part is an exact quantity and we apply the functions `Re` or `Im`, the "exactness" stays unchanged.

```
Re[23/45 + 7.89 I]

Im[23/45 + 7.89 I]
```

Also, note the jump in the phase angle of `Arg` at π.

```
Arg[-1 + 10.0^-10 I]

Arg[-1 - 10.0^-10 I]
```

We look at this graphically (the vertical jump at $arg = \pi$ is a result of `Plot`, which assumes a continuous curve).

```
Plot[Arg[Exp[I φ]], {φ, 0, 2Pi},
     AxesLabel -> {StyleForm[StandardForm[φ]],
                   StyleForm[StandardForm[Arg[Exp[I φ]]]]},
     PlotStyle -> Thickness[0.01]]
```

For real (head `Real`), integer (head `Integer`), and rational (head `Rational`) arguments x, `Re`, `Im`, `Abs`, and `Arg` give the same result as `Complex[`x`, 0]`.

```
Abs[3]

Re[-1]

Im[43/67]

Arg[12]
```

For symbolic arguments, `Re`, `Im`, `Abs`, and `Arg` do not "alter" the input. *Mathematica* does not make any assumptions about the variables `purelyReal` and `purelyImaginary`. With the exceptions of a very few functions, every (non-built-in) symbol is assumed to be a generic complex-valued variable.

```
Re[purelyReal + I  purelyImaginary]

Abs[real^2 + imaginary^2]
```

Using the function `ComplexExpand` (discussed in Chapter 1 of the Symbolics volume [67✱]), expressions containing `Re`, `Im`, `Abs`, and `Arg` can be "simplified" under the assumptions that the involved variables are purely real.

```
ComplexExpand[%]
```

Here is a look at the shape of the absolute value of the arccos function on the complex plane.

```
Plot3D[Abs[ArcCos[x + I y]], {x, -Pi, Pi}, {y, -Pi, Pi},
       AxesLabel -> {x, I y, None}, PlotPoints -> 30]
```

It is clearly nondifferentiable across the negative real axis. If we look only at the negative part of the imaginary part, the nondifferentiable is even more visible. (The vertical piece of the surface is a result of `Plot3D`; a more correct version of the picture should not have these pieces.)

```
Plot3D[-Im[ArcCos[x + I y]], {x, -Pi, Pi}, {y, -Pi, Pi},
       AxesLabel -> {x, I y, None}, PlotPoints -> 30]
```

The reason for this discontinuity is explained in the following section.

Mathematical Remark: Branch Points and Branch Cuts of Analytic Functions

To make the inverse functions corresponding to multivalued complex functions unique, we need several copies (2, 3, …, ∞, depending on the function) of the complex plane that are suitably cut open and glued together along branch cuts. The starting, and ending points of the branch cuts are (typically) the branch points. The resulting multivalued (multisheeted) surface is called a Riemann surface. (We come back to Riemann surfaces repeatedly throughout the *GuideBooks*.) For the numerical computation of these functions, the built-in versions of the *Mathematica* functions always stay on one branch (sheet) of the Riemann surface. If one moves along a path on such a sheet, all functions are continuous (assuming the path does not hit a pole or a singularity). But when crossing a branch cut, the value of the function jumps discontinuously. Function values on higher or lower branches are usually different, often being the conjugates of each other or differ by fixed values. See any textbook on function theory or applied mathematics (e.g., [57✱], [1✱], [13✱], [32✱], [36✱], [39✱], [25✱], and [24✱]). For a detailed listing of the branch cuts of all *Mathematica* functions see, http://functions.wolfram.com.

The branch cuts in the complex planes are different for the various functions (no mathematical theorem for how to make the cuts exists, but there are some conventions). The cuts for the functions introduced above are as follows:

`Sqrt[z]`	$(-\infty, 0)$
`z^s`	$(-\infty, 0)$ for $\mathrm{Re}(s) > 0$ and s not an integer
	$(-\infty, 0]$ for $\mathrm{Re}(s) < 0$ and s not an integer
`ArcSin[z]`	$(-\infty, -1)$ and $(1, \infty)$
`ArcCos[z]`	$(-\infty, -1)$ and $(1, \infty)$
`ArcTan[z]`	$(-i\infty, -i)$ and $(i, i\infty)$
`ArcCot[z]`	$[-i, i]$
`ArcSec[z]`	$(-1, 1)$
`ArcCsc[z]`	$(-1, 1)$
`ArcSinh[z]`	$(-i\infty, -i)$ and $(i, i\infty)$
`ArcCosh[z]`	$(-\infty, 1)$
`ArcTanh[z]`	$(-\infty, -1]$ and $[1, \infty)$
`ArcCoth[z]`	$[-1, 1]$
`ArcSech[z]`	$(-\infty, -1]$ and $(1, \infty)$
`ArcCsch[z]`	$(-i, i)$
`Arg[z]`	$(-\infty, 0)$

(The discontinuity of `Arg` as a function of a complex variable is of different nature because it is not an analytic function.) For most applications these choices of the branch cuts is convenient, but sometimes other branch cut positions are preferable [27✱].

The branch cuts of the inverse trigonometric functions follow largely [18✱] from the branch cuts of the `Log` and `Power`. (And because of the identities $z^\alpha = e^{\alpha \ln(z)}$ and $\ln(z) = \lim_{\alpha \to 0} \alpha\,(z^\alpha - 1)$, the branch cut of the logarithm function and the power function should coincide.) Every inverse trigonometric function can be expressed as a composition of logarithms and square roots [18✱]. (The function `TrigToExp` rewrites inverse trigonometric functions in the more basic functions `Log` and `Power`.)

```
Map[# == TrigToExp[#]&,
    {ArcSin[ξ], ArcCos[ξ], ArcTan[ξ],
     ArcCot[ξ], ArcSec[ξ], ArcCsc[ξ]}] // TableForm
```

Also, the inverse hyperbolic function can be expressed using `Log` and `Power`.

```
Map[# == TrigToExp[#]&,
    {ArcSinh[ξ], ArcCosh[ξ], ArcTanh[ξ],
     ArcCoth[ξ], ArcSech[ξ], ArcCsch[ξ]}] // TableForm
```

Even the ordinary exponentiation is for noninteger powers not unique. Here, the cut is along $(-\infty, 0)$.

```
Plot3D[Im[(x + I y)^(1/3)], {x, -2, 2}, {y, -2, 2},
       PlotPoints -> 30,
       AxesLabel -> {StyleForm[StandardForm[x]],
                     StyleForm[StandardForm[y]], None}]
```

If we follow `Im[(x + I y)^(1/3)] = Im[z^(1/3)] = Im[`$|z|\, e^{i \arg(z)/3}$`]` along the unit circle, after one cycle, we do not get back to the original value. This happens only after the third cycle. We can clearly see in this picture that when starting at $\sqrt[3]{-1}$ and going around the circle of radius 1, the value $\sqrt[3]{-1}$ is *not* -1 on the same sheet of the Riemann surface. In the following picture, we show the real part of $\exp(2\pi i t/3)$ as a function of t.

```
Plot[Im[(Exp[I 2 Pi t/3])], {t, 0, 4}, AxesLabel ->
     (StyleForm[StandardForm[#]]& /@ {t, "Im[Exp[2 I Pi t/3]]"})]
```

The logarithm is also not unique; again, we cut along the negative real axis.

```
Plot3D[Im[Log[x + I y]], {x, -2, 2}, {y, -2, 2},
       PlotPoints -> 40, AxesLabel -> {StyleForm[StandardForm[x]],
       StyleForm[StandardForm[y]], None}]
```

In addition to $e^0 = 1 \to \ln(1) = 0$, we also have the following: $e^{k\, 2\pi i} = 1,\ k \in \mathbb{Z}$.

```
Exp[4 Pi I]
```

Be aware that composite functions have their branch cuts uniquely determined by their constituent functions. Take, for example, the function $f(z) = \sqrt{z^2 - 1}$ in the complex z-plane. The two branch points are $z_{\text{bp}} = \pm 1$ and the branch cut is typically chosen as the straight line connecting these two branch points. But the `Sqrt` function will have a branch cut whenever its argument is negative. For the argument $z^2 - 1$, this is the case for real z in the interval $-1 < z < 1$ *and* for all z on the imaginary axis. This fact explains the look of the following picture.

```
Plot3D[Im[Sqrt[(x + I y)^2 - 1]], {x, -2, 2}, {y, -2, 2},
       PlotPoints -> 50]
```

In the last picture, the "branch cut" along the imaginary axis does not connect any branch points; instead it forms a discontinuity in form of a closed loop running from $i\infty$ to $-i\infty$.

The process of building all elementary and special functions from addition, multiplication, and, at the end, the power function (or the logarithm function) yields consistent, but compared with textbook practice, sometimes unusual results for numerical values of composite functions. Here is a simple example: The function $\ln(-\exp(i \arccos(z)))$ (this function can also be written as $\ln\left(-z - i\left(1 - z^2\right)^{1/2}\right)$. On the branch cuts $(-\infty, -1]$ and $[1, \infty)$ of arccos, the function values are of the form $\pi - i\,y$ and $i\,y$ with purely real positive y. As a result, the function values of $\exp(i \arccos(z))$ are purely real positive on the interval $[1, \infty)$. So $-\exp(i \arccos(z))$ is negative on $[1, \infty)$ and continuous from below. Because this expression is negative, it experiences the branch cut of the outer logarithm. With $\ln(-z)$ being continuous from above, the values of the expression $\ln(-\exp(i \arccos(z)))$ do not agree with either of the limits from below or above. The following two graphics show the real and imaginary parts of $\ln(-\exp(i \arccos(z)))$. We use high-precision arithmetic to calculate the function values.

```
ƒStrange[z_] := Log[-Exp[I ArcCos[z]]]

(* high-precision function values *)
fValues = Table[{x, y, N[ƒStrange[x + I y], 22]},
                {x, -3, 3, 6/32}, {y, -3, 3, 6/32}];

(* form polygons from points *)
polygons = Table[Polygon[{#[[i, j]], #[[i + 1, j]],
                          #[[i + 1, j + 1]], #[[i, j + 1]]}],
                 {i, Length[#] - 1}, {j, Length[#[[1]]] - 1}]&[fValues];

(* show real and imaginary part *)
Show[GraphicsArray[Function[reIm,
Graphics3D[{EdgeForm[], Map[MapAt[reIm, #, 3]&, polygons, {4}]},
                  BoxRatios -> {1, 1, 0.6}, Axes -> True,
                  AxesLabel -> {"x", "y", None},
                  PlotLabel -> reIm]] /@ {Re, Im}]]
```

The vertical wall along the interval $[1, \infty)$ might seem strange in the first moment, but follows uniquely from a consistent and fixed branch cut structure of all elementary functions.

Another sometimes encountered pair of functions that differ only on a line segment is $\operatorname{arccoth}(z)$ and $\ln\left((z+1)^{1/2} \big/ (z-1)^{1/2}\right)$. (The last form one typically obtains by solving $\coth(w) = z$ with respect to w after expressing $\coth(w)$ through exponentials.) These two functions differ on the interval $-1 < \operatorname{Re}(z) < 0$, $\operatorname{Im}(z) = 0$ by $i\,\pi$.

By composing functions with branch cuts, one can obtain quite complicated branch cut structures. For a preliminary attempt to calculate them in a programmatic way, see [10✱], [22✱], and [6✱].

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

■ 2.2.6 Do Not Be Disappointed

We discuss a few things for which *Mathematica*, as well as several other computer algebra systems, are frequently and unfairly criticized. *Mathematica* has no explicit type declaration for variables, and so every symbolic quantity is considered to be able to assume a general complex value. This assumption has the effect that a variety of expressions that could be simplified for real numbers, are no longer simplified with complex numbers. (This subsection closely follows [64✱]; see also [4✱], [53✱], [59✱], [43✱], [19✱], [11✱], [20✱], [21✱], [5✱], [16✱], and the early work [15✱].)

Mathematica does not simplify a number of expressions that one initially thinks could be simplified. Known rules for positive real numbers often do not hold for arbitrary complex numbers and every variable is assumed to be a generic complex quantity.

Mathematica does not recognize the following simplifications, which are correct for positive real arguments.

■ $\sqrt{u}\,\sqrt{v} = \sqrt{u\,v}$:

```
Sqrt[u] Sqrt[v]
```

■ $\sqrt{u^2} = u$:

```
Sqrt[u^2]
```

■ $\sqrt{1/u} = 1\big/\sqrt{u}$:

```
Sqrt[1/u]
```

■ $\sqrt{e^x} = e^{x/2}$:

```
Sqrt[Exp[x]]
```

■ $\ln(u\, v) = \ln u + \ln v$:

```
Log[u v]
```

■ $\ln(u^2) = 2 \ln u$:

```
Log[u^2]
```

■ $\ln(1/u) = -\ln u$:

```
Log[1/u]
```

To its credit, *Mathematica* does not recognize the following "simplifications".

■ $\sqrt{u}\ \sqrt{v} = \sqrt{u\, v}$:

$$u = -1 \quad v = -1 \quad \rightarrow \sqrt{-1}\ \sqrt{-1} = i^2 = -1 \neq \sqrt{(-1)\,(-1)} = \sqrt{1} = 1$$

```
u = -1; v = -1;
{Sqrt[u] Sqrt[v], Sqrt[u v]}
```

■ $\sqrt{u^2} = u$:

$$u = -1 \rightarrow \sqrt{\left((-1)^2\right)} = 1 \neq -1$$

```
u = -1;
{Sqrt[u^2], u}
```

■ $\sqrt{\frac{1}{u}} = \frac{1}{\sqrt{u}}$:

$$u = -1 \rightarrow \sqrt{\frac{1}{-1}} = \sqrt{-1} = i \neq \frac{1}{\sqrt{-1}} = \frac{1}{i} = -i$$

```
u = -1;
{Sqrt[1/u], 1/Sqrt[u]}
```

■ $\sqrt{e^x} = e^{x/2}$:

$$x = 2\,\pi\, i \rightarrow \sqrt{e^{2\,\pi\, i}} = \sqrt{1} = 1 \neq \left(e^{(2\,\pi\, i)/2}\right) = (e^{\pi}\, i) = -1$$

```
x = 2 I Pi;
{Sqrt[Exp[x]], Exp[x/2]}
```

■ $\ln(u\, v) = \ln u + \ln v$:

$$u = -1 \;\; v = -1 \rightarrow \ln((-1)\,(-1)) = \ln(1) = 0 \neq \ln(-1) + \ln(-1) = \pi\, i + \pi\, i = 2\,\pi\, i$$

Note that $\ln(-1) = i\,\pi$ because $e^{i\pi} = -1$.

```
u = -1; v = -1;
{Log[u v], Log[u] + Log[v]}
```

■ $\ln(u^2) = 2\ln u$:

$$u = -1 \to \ln((-1)^2) = \ln(1) = 0 \neq 2\ln(-1) = 2\,\pi\,i$$

```
u = -1;
{Log[u^2], 2 Log[u]}
```

■ $\ln(1/u) = -\ln u$:

$$u = -1 \to \ln\left(\frac{1}{-1}\right) = \ln(-1) = \pi\,i \neq -\ln(-1) = -\pi\,i$$

```
u = -1;
{Log[1/u], -Log[u]}
```

Sometimes, we would nevertheless wish *Mathematica* to use the above-described rules—this can be forced with the function `PowerExpand`, which is discussed in Chapter 1 of the Symbolics volume [67✱]. Also, by giving *Mathematica* additional information about the domain of variables more simplifications become possible. Here are two simple examples (we will discuss the function `Simplify` in detail in Chapter 1 of the Symbolics volume [67✱]).

```
Simplify[Sqrt[u] Sqrt[v], And[u > 0, v > 0]]

Simplify[Sqrt[u^2], Element[u, Reals]]
```

Note that branch cut problems are not affected by the above listing. As mentioned, the reader might get something different from x in *inverseFunction*[*function*[x]].

```
x = 3 Pi I; {Log[Exp[x]], x}
```

In the last example, only the main branch is used for the logarithm (because `Exp[x]` is computed first, and then `Log[-1]`). But as a multivalued function, we have $\ln(z) = \ln(z)_H + k\,2\,\pi\,i$, with k an arbitrary integer; $\ln(z)_H$ means the value on the main branch.

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

■ 2.2.7 Exact and Approximate Numeric Quantities

Although mathematical constants such as e (`E`), π (`Pi`), the golden ratio (`GoldenRatio`), and degree (`Degree`) have the head `Symbol`, they nevertheless represent numerical quantities. It is sometimes necessary to convert them to approximate numbers. This conversion can be done with the command `N`.

`N[`*toBeNumericalized*`,` *numberOfDigits*`]`

computes the numerical value of *toBeNumericalized* to *numberOfDigits* digits. If the second argument is left out, or is the symbol `MachinePrecision`, the computations are done with machine accuracy, usually 16 to 19 digits, depending on the hardware.

The input $\exp(10^{-100})$ stays unevaluated. *Mathematica* has no built-in rule to transform this expression.

```
Exp[10^-100]
```

Here is $\exp(10^{-100})$ computed to 800 digits. The result clearly shows the contributions of the first few terms of the Taylor series expansion.

```
N[Exp[10^-100], 800]
```

The next two inputs calculate a machine number result for the difference of the last number to 1.

```
N[% - 1]
```

```
N[%% - 1, MachinePrecision]
```

First, the difference is calculated, then `N` is applied to convert the result to a machine number. If we would first convert the high-precision number to a machine number and then subtract the 1, the result would be 0. because after conversion to a machine number we have only about 16 digits.

```
N[%%%] - 1
```

Because calculations with symbolic expressions are typically much slower and more memory-intensive than with approximate numbers, whenever possible, `N[… ]` should be used (e.g., in self-constructed graphics). However, the loss of precision may generate misleading results, particularly with machine-precision computations.

If a decimal number with *n* digits is input, *Mathematica* assumes that only these *n* digits are correct. If *n* is less than machine precision, all remaining digits (up to machine precision) are interpreted as decimal zeros. If *n* is greater than machine precision, any digits not given explicitly are assumed to be indefinite. Given a number with *n* (*n* less than machine precision) digits, it is a bit more involved to get a new number with *m* digits with $m > n$. (We discuss how to do this in great detail in Chapter 1 of the Numerics volume [66✱].) Thus, for example, the following, does not work.

```
N[Sin[2.00], 40]
```

The inner `Sin[2.00]` evaluated to a machine precision.

```
Sin[2.00]
```

```
FullForm[Sin[2.00]]
```

Moreover, trailing zeros are not displayed.

```
InputForm[2.0]
```

To get a result with a lot of digits, we have to give input with that many digits.

```
N[Sin[2.00000000000000000000000000000000000000000000000000\
        00000000000000000000000000000000000000000000000000\
        00000000000000000000000000000000000000000000000000\
        00000000000000000000000000000000000000000000000000], 200]
```

In the last input, the 200 is not necessary. *Mathematica* will compute the expression to the precision that is justified by the precision of the input.

```
Sin[2.00000000000000000000000000000000000000000000000000\
      00000000000000000000000000000000000000000000000000\
      00000000000000000000000000000000000000000000000000\
      00000000000000000000000000000000000000000000000000]
```

Here are shorter forms of the last input.

```
Sin[N[2, 200]]
```

```
Sin[2.`200]
```

A number that is zero to *n* digits can be input as `0``n`. (This means $|0``n\,| \leq 10^{-n-1}$.)

```
0``100
```

Next, we calculate arccot(*zero*) for three different *zero*s.

```
ArcCot[0]
```

```
ArcCot[0.0]
```

```
ArcCot[0``50]
```

```
% - Pi/2
```

`N[`*expr*`, `*prec*`]` calculates expression to precision *prec*. For most cases, this means that the result has *prec* digits. If the result is a complex number with real and imaginary parts of very different size, the smaller (in magnitude) part might have fewer digits. Here is an example.

```
expr = (100 Pi -
 19132026092227517122744933006259318953191397092415777/
  5555080271647593936029563103709759827268790222640350 I)^
                               (10 GoldenRatio + I EulerGamma)
```

`N[`*expr*`, 50]` gives the real part to 50 correct digits. But not a single validated digit of the imaginary part could be found.

```
N[expr, 50]
```

`N[`*expr*`, 100]` gives just three validated digits for the imaginary part and shows that the imaginary part is more than 100 orders of magnitude smaller than the real part.

```
N[expr, 100]
```

The following input calculates 20 validated digits for the imaginary part.

```
$MaxExtraPrecision = 1000;
N[Im[expr], 20]
```

Here is a sum of 11 cosines.

```
cosSum10 = (6 - 15 Cos[1] + 27 Cos[2] + 9 Cos[3] -
            6 Cos[4] + 45 Cos[5] + 16 Cos[6] + 20 Cos[7] -
            5 Cos[8] + 6 Cos[9] + 24 Cos[10]);
```

Using machine precision, the sum evaluates to a small nonzero value in the order of 10^{-n+1} where n denotes the number of digits used for machine arithmetic (the 1 in the exponents stems from the fact that the coefficients are of order 10^1).

```
N[cosSum10]
```

Using high-precision arithmetic, we get the correct answer.

```
N[cosSum10, 20]
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

2.3 Nested Expressions

■ 2.3.1 An Example

The expression

$$\ln(x^2 + 5\,x) + \sin(4\,t^2\,y^4) - t\,y^{2+\exp(-3\,x)}$$

is nested a couple of times, as can be seen from the following.

```
Log[x^2 + 5 x] + Sin[4 t^2 y^4] - (t y^(2 + Exp[-3 x]))
```

A mathematically insignificant, but technically very important, difference exists between the input and the output of the last expression.

> *Mathematica* writes all expressions in a canonical ordered form, which makes it *much* easier to compare and sum various expressions. (Most expressions are not transformed in a canonical mathematical form; this would be too expensive.)

The `FullForm` and `TreeForm` of the above expression are both a bit complicated.

```
FullForm[%]

TreeForm[%%]
```

Because we want to work with this expression later, we give it the name `expression`.

```
expression = %%%
```

Here is its head.

```
Head[expression]
```

Now, we get an overview of the structure of larger expressions.

```
Short[expression^expression^expression]

Shallow[FullForm[expression^expression^expression]]
```

The two functions `Short` and `Shallow` work as follows.

`Short[`*expression*`]`

writes *expression* in a shorter form (that is typically one line long).

`Shallow[`*expression*`]`

writes *expression* in skeleton form.

The result of `Shallow[FullForm[`*expression*`]]` involved a `Skeleton`.

`Skeleton[`*n*`]`

represents a sequence of *n* omitted elements in an expression printed out with `Short` or `Shallow`. The short form is displayed as << *n* >>. The input form of the expression containing << *n* >> stays unchanged.

Both `Short` and `Shallow` allow the input of a second argument.

`Short[`*expression*`, `*n*`]`

writes *expression* in shorter form, using at most *n* rows.

`Shallow[`*expression*`, `*n*`]`

writes *expression* in shorter form, where all partial expressions having a depth greater than *n* are written in skeleton form.

We will come back to the precise definition of the word “depth” in a moment. Here is a larger set of numbers (the semicolon prevents any printing).

```
table = Table[i, {i, 1000}];
```

Here is a short form consisting of three rows.

```
Short[table  // OutputForm, 4]
```

For comparison, here is `Shallow[table]`.

```
Shallow[table]
```

We now look at the effect of the second argument of `Shallow` on `expression`.

```
Shallow[expression, 1]

Shallow[expression, 2]

Shallow[expression, 3]

Shallow[expression, 4]

Shallow[expression, 5]

Shallow[expression, 6]

Shallow[expression, 7]
```

And starting from $n = 8$, we recover the whole expression.

```
Shallow[expression, 8]

Shallow[expression, 9]
```

The next input uses a nested `Shallow`.

```
Shallow[Shallow[expression, 5], 4]
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

■ 2.3.2 Analysis of a Nested Expression

In this subsection, we discuss the most important tools for analyzing the structure of large (often extremely large) *Mathematica* expressions. In solving "real-life" problems with *Mathematica*, expressions may require several megabytes or sometimes several tens or even hundreds of megabytes. It is immediately obvious that looking at a `FullForm` and/or `TreeForm` taking several dozen to several hundred pages (graphics, matrices, integrals of complicated functions, recursive functions, etc.) is of no use. Here is a really big expression.

```
veryBigExpression = TreeForm[Nest[Function[x,
                    Sin[ξ x + Exp[1/Log[Sqrt[Tan[x^(2 ⅈ)]]]]]], x, 10]];
```

Its tree form is amusing, but practically useless. The overall shape of the tree form is hard to grasp, and the details are virtually invisible. The next little program (to be made active using the Make Input button) will generate a notebook with the tree form of `veryBigExpression`.

```
NotebookPut[Notebook[{Cell[BoxData[
    MakeBoxes[#, StandardForm]], CellHorizontalScrolling -> True,
              FontColor -> RGBColor[1, 0, 0], FontSize -> 9]},
     ScrollingOptions  -> {"HorizontalScrollRange" -> 500000},
     WindowSize -> {500, 600}, WindowFrameElements -> {"CloseBox"},
     WindowMargins -> {{0, 0}, {Automatic, 10}},
     Background -> GrayLevel[0]]]&[(* the big expression *) veryBigExpression]
```

The following graphic shows an outline of the tree form of `veryBigExpression` (just look at the graphic, the details of the programming will be discussed later). The tree has more than 17000 roots, and the deepest roots extend over more than 80 levels.

```
ListPlot[-Length[First[#]]& /@ Cases[
         MapIndexed[C[#2]&, veryBigExpression, {-1}, Heads -> True],
                        _C, {0, Infinity}, Heads -> True],
         Frame -> True, Axes -> False, PlotStyle -> {PointSize[0.002]}]
```

(Depending on the actual settings, `TreeForm` might not accept such expressions, but instead generates error messages such as `Format::lcont: ... or Format::toobig: ...`. In such cases, even `Short` and `Shallow` are of limited use, because the structure of parts deep inside is not accessible.

Here is a rather complicated expression (small compared with `veryBigExpression`, but large enough for the following analysis; the example from Subsection 2.3.1, with two additional terms).

$$\ln(x^2 + 5x) + \sin(4t^2 y^4) - t\, y^{2+\exp(-3x)} + 45\, t^6 - 4$$

We call it `expression2`.

```
expression2 = Log[x^2 + 5 x] + Sin[4 t^2 y^4] -
              (t y^(2 + Exp[-3 x])) + 45 t^6 - 4
```

Again, it is reordered into a canonical normal form. The `FullForm` of `expression2` is quite big.

```
FullForm[expression2]
```

The `TreeForm` is already hard to read (at least if the lines have to be broken).

```
TreeForm[expression2]
```

Using the function `Part`, we can decompose `expression2` (and every other expression).

`Part[`*expression*`,` *i*`]` or *expression*`[[`*i*`]]`
gives the *i*th part of *expression*. *expression*`[[0]]` gives the head of *expression*.

The *i*th part ($i > 0$) of an expression *expr* can be viewed as the *i*th argument of `Head[`*expr*`]`. We illustrate the formation of the various parts of an expression by looking at `expression2`.

```
expression2[[0]]

expression2[[1]]

expression2[[2]]

expression2[[3]]

expression2[[4]]

expression2[[5]]
```

Because `expression2` has only five parts, the input `expression2[[6]]` gives a message.

```
expression2[[6]]
```

If we want to further decompose the parts we already obtained, we can use `Part` on the already extracted parts again, or more conveniently, one of the following alternatives.

`Part[`*expression*`, `*i*`, `*j*`, …]`
or
expression`[[`*i*`]][[`*j*`]][[…]]…[[…]]`
or
expression`[[`*i*`, `*j*`, …]]`
gives the … part of the … parts of the *j*th part of the *i*th part of *expression*. This is equivalent to `Part[…[Part[Part[`*expression*`, `*i*`], `*j*`]…], …]`.

Here is the second part of `expression2` in detail.

```
FullForm[expression2[[2]]]
```

Here are its two subparts.

```
expression2[[2, 1]]
```

```
expression2[[2, 2]]
```

For long expressions, it may be more convenient to count from the end.

`Part[`*expression*`, −`*i*`]` or *expression*`[[−`*i*`]]`
gives the *i*th part of *expression*, counting from the end of *expression*.

We now extract the parts of `expression2`, starting at the end.

```
expression2[[-1]]
```

```
expression2[[-2]]
```

```
expression2[[-3]]
```

Positive and negative indices can be arbitrarily mixed. Here, we take the minus second part of the (plus) second part.

```
expression2[[2, -2]]
```

This input extracts the same subexpression.

```
expression2[[-4, 1]]
```

If the second element of `Part` is a list, these parts are returned.

```
expression2[[{4, 5}]]
```

Besides explicit integers, the command `All` can be used to specify parts. The following input takes the fourth and fifth elements of `expression2`. The following input first takes the fourth and fifth element of `expression2`, and then takes the second element of all of its subparts.

```
expression2[[{4, 5}, All, 2]]
```

How many indices are needed to completely decompose an expression? The answer to this question is provided by the function `Depth`. (This is what we were referring to in Subsection 2.3.1 when we used the word depth.)

`Depth[`*expression*`]`

gives *indices* + 1, where *indices* is the number of indices needed to uniquely specify any part (obtained with `Part` with a nonleading zero) of *expression* (the +1 results from the head).

For `expression2`, we require 7 – 1 = 6 indices.

```
Depth[expression2]
```

If we analyze the third part of `expression2` further, it becomes obvious that indeed six indices are needed for a unique specification of its parts.

```
expression2[[3]]

expression2[[3, 3]]

expression2[[3, 3, 2]]

expression2[[3, 3, 2, 2]]

expression2[[3, 3, 2, 2, 2]]
```

And now six indices are needed.

```
expression2[[3, 3, 2, 2, 2, 2]]
```

Be aware of the nonzero restriction for the part specification. Here is an expression with a more complicated head than argument.

```
complicatedExpression =
head1[subHead1[subSubHead1[0], subSubHead2[subSubSubHead1[Ψ]]]][
                                          argument1[subArgument1[2]]];
```

The depth of the expression is 4.

```
Depth[complicatedExpression]
```

We need 4 – 1 = 3 integers to specify the position of the `2` in `complicatedExpression`.

```
complicatedExpression[[1, 1, 1]]
```

The position of `Ψ` in `complicatedExpression` is specified by five integers. But `Ψ` appears in the head (leading `0`), so `Depth` does not take the head into account.

```
complicatedExpression[[0, 1, 2, 1, 1]]
```

Let us deal now with some other examples using the functionality of `Part`. `Λ` is a nested *Mathematica* expression. The λi indicate the level *i*.

```
Λ = λ0[λ1[λ2[λ3[1, 1, 1], λ3[1, 1, 2], λ3[1, 1, 3]],
          λ2[λ3[1, 2, 1], λ3[1, 2, 2], λ3[1, 2, 3]],
          λ2[λ3[1, 3, 1], λ3[1, 3, 2], λ3[1, 3, 3]]],
       λ1[λ2[λ3[2, 1, 1], λ3[2, 1, 2], λ3[2, 1, 3]],
          λ2[λ3[2, 2, 1], λ3[2, 2, 2], λ3[2, 2, 3]],
          λ2[λ3[2, 3, 1], λ3[2, 3, 2], λ3[2, 3, 3]]],
       λ1[λ2[λ3[3, 1, 1], λ3[3, 1, 2], λ3[3, 1, 3]],
          λ2[λ3[3, 2, 1], λ3[3, 2, 2], λ3[3, 2, 3]],
          λ2[λ3[3, 3, 1], λ3[3, 3, 2], λ3[3, 3, 3]]]];
```

Here are its first, second, and third parts.

```
Λ[[1]]
```

```
Λ[[2]]
```

```
Λ[[3]]
```

The next example keeps all parts at level 1. The head of the resulting expressions is the same as the head of the original expression.

```
Λ[[{1, 2, 3}]]
```

Instead of explicitly specifying all parts, we can also use `All`.

```
Λ[[All]]
```

Now, the second element of all parts is extracted.

```
Λ[[All, 2]]
```

This input is an equivalent formulation.

```
Λ[[{1, 2, 3}, 2]]
```

Here, the third element of all elements at level three is selected.

```
Λ[[All, All, 3]]
```

```
Λ[[{1, 2, 3}, {1, 2, 3}, 3]]
```

The next input selects all first elements from all first elements from all elements of Λ.

```
Λ[[All, 1, 1]]
```

Now, we take the first element of all elements of the first element of all elements.

```
Λ[[All, 1, All, 1]]
```

This process selects all heads from all elements at level 1.

```
Λ[[All, 0]]
```

This process selects all heads from all elements at level 2.

```
Λ[[All, All, 0]]
```

Here is another example. `poly` is a polynomial in `x`.

```
poly = (x^2 c[2] + x^3 c[3] + x^4 c[4] + x^5 c[5])
```

This process extracts the powers of `x`. The head of the resulting expression is now `Plus`.

```
poly[[All, 1]]
```

This process extracts the constants `c[`*i*`]`.

```
poly[[All, 2]]
```

The first parts of all terms of the form `Power[x, `*n*`]` are just `x`. After extraction, we have `x + x + x + x`, which evaluates to `4 x`.

```
poly[[All, 1, 1]]
```

The second part of the first part of all terms of the form `Power[x, `*n*`]` is just *n*. After extraction, we have `2 + 3 + 4 + 5`, which evaluates to `14`.

```
poly[[All, 1, 2]]
```

The zeroth part of the first part of all terms of the form `Power[x, `*n*`]` is the head `Power`. After extraction, we have `Power+Power+Power+Power`, which evaluates to `4 Power`.

```
poly[[All, 1, 0]]
```

The first part of the second parts of all terms of the form `c[`*n*`]` is *n*. After extraction, we have `2+3+4+5`, which evaluates again to `14`.

```
poly[[All, 2, 1]]
```

This input reproduces the original polynomial. In each step, we took all elements.

```
poly[[All, All, All]]
```

The depth of `poly` is 4, which means all elements can be extracted with a three-element `Part` specification. As a result, the following input generates messages.

```
poly[[All, All, All, All]]
```

Using the head `List` in the second argument of `Part`, extracts the specified parts and applies the head of the original expression to the result. In the next example, the second and third term of the polynomial `poly` is extracted.

```
poly[[{1, 2}]]
```

This yields the sum of three copies of the first summand.

```
poly[[{1, 1, 1}]]
```

Often, it is equally important to answer the reverse formulation of the question: which indices correspond to a certain (prescribed) part of an expression? The answer to this question is given by `Position`. The following finds the position of `x`, `y`, and `2` in `expression2` (`x` appears three times).

```
Position[expression2, x]

{expression2[[3, 3, 2, 2, 2, 2]],
 expression2[[4, 1, 1, 2]],
 expression2[[4, 1, 2, 1]]}
```

`y` appears twice.

```
Position[expression2, y]

{expression2[[3, 3, 1]], expression2[[5, 1, 3, 1]]}
```

`2` appears three times.

```
Position[expression2, 2]

{expression2[[3, 3, 2, 1]], expression2[[4, 1, 2, 2]],
 expression2[[5, 1, 2, 2]]}
```

The composite expression `t^2` appears only once.

```
Position[expression2, t^2]

{expression2[[5, 1, 2]]}
```

`Position[`*expression*`,` *subExpression*`]`

gives a list $\{i_1, i_2, \ldots, i_n\}$ of the indices needed to extract *subExpression* from *expression* using `Part`, where *expression*`[[`$i_1, i_2, \ldots\ i_n$`]]` is exactly *subExpression*. If a *subExpression* appears more than once, all positions of *subExpression* in *expression* are included in a list of the type `{`$position_1$`,` $position_2$`,` …`,` $position_n$`}`, each $position_i$ of the form $\{positionAtLevel1_{j_1}, positionAtLevel2_{j_2}, \ldots, positionAtLeveln_{j_n}\}$.

Here are two more examples. In the expression `1[1]`, we have two “`1`”s. One as a head (position 0), and one as the first argument.

```
Position[1[1], 1, {0, Infinity}, Heads -> True]
```

When the expression one is looking for is the whole expression, the result of `Position` is `{{}}`.

```
Position[1, 1, {0, Infinity}, Heads -> True]
```

```
Position[1, 1, {0, Infinity}]
```

If a *subExpression* does not exist at the specified level, the result is the empty list `{}`.

```
Position[1, 1, {1, Infinity}]
```

The position of *expr* with *expr* is `{}` (zero indices are needed to describe the position of expression itself). So the next input returns `{{}}`.

```
Position[1, 1]
```

Frequently, we are interested not only in a particular part of an expression, but also in all parts at a prescribed level.

`Level[`*expression*`,` *levelSpecification*`]`

gives all parts of *expression*, which have indices at level *levelSpecification*.

Definition: `Level`
Level n ($n > 0$, integer) of an expression is the set consisting of all subexpressions of the expression whose elements require exactly n indices to be identified or selected using `Part`. Level n ($n < 0$, integer) of an expression is the set of all subexpressions of the expression that have depth exactly n (as defined by `Depth`). Level 0 of an expression is the expression itself.

The level specifications of Level are as follows:

`0`

expression itself

i

levels 1 to *i* of *expression*

`Infinity`

all levels (if any exist), excluding *expression* itself

`{0, Infinity}`

all levels (if any exist), including *expression* itself

`{`*i*`}`

only level *i* of *expression*

`{-1}`

the lowest level (“root” of the `TreeForm`) of *expression*

`{`i_1, i_2`}`

levels i_1 to i_2 of *expression* (this means all levels that are not above i_1 and not below i_2)

Here are all expressions at the first level of `expression2`.

```
Level[expression2, 1]
```

Here are those at levels 0 and 1. (`expression2` itself is included.)

```
Level[expression2, {0, 1}]
```

Here is `expression2` itself.

```
Level[expression2, {0}]
```

Now, we show all expressions up to level 2 (starting from level 1 on).

```
Level[expression2, 2]
```

Here are the ones exactly at level 2.

```
Level[expression2, {2}]
```

By the definition of `Level`, these expressions should be all parts of `expression2` that can be extracted using `expression2[[`*i*, *j*`]]` (two arguments).

```
{expression2[[2, 1]], expression2[[2, 2]], expression2[[3, 1]],
 expression2[[3, 2]], expression2[[3, 3]], expression2[[4, 1]],
 expression2[[5, 1]]}
```

Now, here is level `{3}`.

```
Level[expression2, {3}]
```

These are just the terms that can be extracted with `Part` using three indices.

```
{expression2[[2, 2, 1]], expression2[[2, 2, 2]], expression2[[3, 3, 1]],
 expression2[[3, 3, 2]], expression2[[4, 1, 1]], expression2[[4, 1, 2]],
 expression2[[5, 1, 1]], expression2[[5, 1, 2]], expression2[[5, 1, 3]]}
```

Now, if we look from below (at the leaves of the tree, if the expression is viewed as a tree), we get all elementary objects.

```
Level[expression2, {-1}]
```

This is because they individually have depth 1.

```
{Depth[-4], Depth[45], Depth[t], Depth[6], Depth[-1],
 Depth[t],  Depth[y],  Depth[2], Depth[E], Depth[-3],
 Depth[x],  Depth[5],  Depth[x], Depth[x], Depth[2],
 Depth[4],  Depth[t],  Depth[2], Depth[y], Depth[4]}
```

Here are the objects in `expression2` with depth 2.

```
Level[expression2, {-2}]
```

With negative indexed levels, we cannot determine anything about the indices needed in `Part`. Only their depth (using `Depth`) is fixed.

```
{Depth[t^6], Depth[-3x], Depth[5x], Depth[x^2], Depth[t^2], Depth[y^4]}
```

```
{expression2[[2, 2]],    expression2[[3, 3, 2, 2, 2]],
 expression2[[4, 1, 1]], expression2[[4, 1, 2]],
 expression2[[5, 1, 2]], expression2[[5, 1, 3]]}
```

At the level `Infinity`, `expression2` has no structure, because it has a finite size and depth.

```
Level[expression2, {Infinity}]
```

For positive integers *j* and *k*, `Level[`*expr*`, {`*k*`}]` is identical to `Level[Level[`*expr*`, {`*j*`}], {`*k* `+ 1 -` *j*`}]` as long as $k + 1 - j$ is also a positive integer. This means that the level specification can be defined and applied recursively. (The +1 in $k + 1 - j$ results from the enclosing `{}` returned by `Level[`*expr*`, {`*j*`}]`.) The next inputs demonstrate this property for the level `{3}` of `expression2`.

```
Level[expression2, {3}]

Level[Level[expression2, {1}], {3}]

Level[Level[expression2, {2}], {2}]

Level[Level[expression2, {3}], {1}]
```

The function `Length` answers the question: How many parts does an expression have?

`Length[`*expression*`]`
gives the number of parts of *expression* at level 1.

Between the depth of an expression and its levels, we have the relation `Depth[`*expr*`] == Depth[Level[`*expr*`, {`*k*`}]+` *k* `- 1` for all $0 \leq k <$`Depth[`*expr*`]`. Here are the lengths of various subexpressions of `expression2`.

```
Length[expression2]

Length[expression2[[1]]]

Length[expression2[[2]]]

Length[expression2[[3]]]
```

Using the functions `Part` and `Length`, we can write the following structural identity for any *Mathematica* expression: *expr* `==` *expr*`[[0]][`*expr*`[[1]],` *expr*`[[2]], …,` *expr*`[[Length[`*expr*`]]]]`.

Now, we use *Mathematica* to systematically study the lengths of all of the parts at all levels of `expression2`. (We later explain how to program such structural investigations.) Here, we give only the expressions for all levels.

```
Do[CellPrint[Cell[TextData[(* ◦ means Mathematica generated text *)
 {"◦ Length of the parts at level: " <> ToString[i]}], "PrintText"]];
   Print[TableForm["Length[" <> ToString[InputForm[#]] <> "] = " <>
         ToString[Length[#]]& /@ (* the various levels *)
           Level[expression2, {i}]]], {i, -7, 6, 1}]
```

To find out how big an expression is, or how many syntactically correct parts it involves, we can use `LeafCount`.

`LeafCount[`*expression*`]`
gives the number of indivisible leaves of *expression* obtained by splitting it into a hierarchical structure.

The count for `expression2` is 36.

```
LeafCount[expression2]
```

Here are many of them.

```
Level[expression2, {-1}]
```

There are 20 pieces.

```
Length[%]
```

The missing 16 pieces are in the heads. If we also want to include the `Heads` of the various levels in pure form (e.g., `Sin`) in the resulting list, we use an option in `Level`. (Options are discussed in Chapter 3.)

`Heads`

is an option for the function `Level`. `Level[`*expression*`,` *levelSpecification*`, Heads -> True]` includes the heads in the list produced by `Level`.

Now, 16 more leaves are present.

```
Level[expression2, {-1}, Heads -> True]
```

```
Length[%]
```

These are all subexpressions of `expression2` (excluding heads and excluding `expression2` itself).

```
Level[expression2, Infinity]
```

Now, `expression2` is also included.

```
Level[expression2, {0, Infinity}]
```

Here are the corresponding levels with the option `Heads -> True`.

```
Level[expression2, Infinity, Heads -> True]
```

```
Level[expression2, {0, Infinity}, Heads -> True]
```

Including all of the heads, we have 52 elements in the last list.

```
Length[%]
```

Using the function `Complement` (to be discussed in Chapter 6), we can extract all heads of the expression `expression2`. (There are many other ways to extract these heads.)

```
Complement[Level[expression2, {-1}, Heads -> True],
           Level[expression2, {-1}, Heads -> False]]
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

2.4 Manipulating Numbers

■ 2.4.1 Parts of Fractions and Complex Numbers

Two exceptional types of expressions exist in which `Part` cannot be used to extract parts of the expression: rational and complex numbers.

```
FullForm[3/5]
```

It might be expected that the 3 in 3/5 could be extracted with `(3/5)[[1]]` and the 5 with `(3/5)[[2]]`. However, this does not work.

```
(3/5)[[1]]
```

Similarly, we could try to extract the 3 in `3 + 5i` with `(3 + 5I)[[1]]` and the 5 with `(3/5)[[2]]`. This also fails.

```
(3 + 5 I)[[2]]
```

The following example works. (Here `i` is lowercase and not a number; it has head `Symbol`.)

```
(3 + 5 i)[[1]]

(3 + 5 i)[[2]]
```

Here is the reason it works.

```
TreeForm[3 + 5 i]
```

A rational or complex number *number* cannot be decomposed using *number*`[[1]]` or *number*`[[2]]`, even though in `FullForm` it has two arguments. They are called atomic, or raw expressions.

For fractions (head `Rational`), we can get what we want using `Numerator` and `Denominator`.

`Numerator[`*fraction*`]`
gives the numerator of the fraction *fraction*.

`Denominator[`*fraction*`]`
gives the denominator of the fraction *fraction*.

```
Numerator[3/7]

Denominator[3/7]
```

The corresponding commands `Re` and `Im` for extracting the real and imaginary parts of a complex function have already been discussed in Subsection 2.2.4.

```
Re[3 + 7 I]

Im[3 + 7 I]
```

Note that `Re` and `Im` only work with expressions that have numerical values. Thus, this yields no result, because nothing is known about the real and imaginary parts of the variable `indefinite` that could possibly take on complex values.

```
Re[(3.9 + 9.7 I) indefinite]
```

However, the appropriate rules are built in for mathematical constants.

```
Re[Pi]

Im[Pi + I]

Abs[GoldenRatio]
```

The analogous statement holds for algebraic numbers. Simple expressions are typically simplified whereas larger ones keep the `Re` or `Im`.

```
Im[Sqrt[2]]

Im[Sqrt[-2]]

Im[Sqrt[2 I] - (-3)^(1/4)]

Re[Sqrt[2 + Sqrt[5]] + Sqrt[3 + 2I]]
```

Here is a more complicated example.

```
Re[(4(-1)^(I + 3 (-1)^I)^(1/3) +
    (I + 3(-1)^I)^(1/3))^(1/4) - (-5)^(1/(1 + 7I))]
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

■ 2.4.2 Digits of Numbers

Sometimes, we need to extract the digits that make up a given integer or real number. The following two functions can be used.

`IntegerDigits[`*integer*`, ` *base*`]`

produces a list of the digits of the integer *integer* relative to the base *base*. If *base* is not present, it is taken to be 10.

`RealDigits[`*realNumber*`, ` *base*`]`

produces a list consisting of the digits making up the real number *realNumber* (head `Real`) in the base *base*, along with the number of digits to the left of the decimal point. If *base* is not present, it is taken to be 10.

Here are a few obvious examples to illustrate the effect of `IntegerDigits`.

```
IntegerDigits[123456789]
```

```
IntegerDigits[-123456789]
```

```
IntegerDigits[1024, 2]
```

```
IntegerDigits[6 222 + 45, 222]
```

If the first argument of `IntegerDigits` is not an integer, an error message is generated and the input is returned unchanged.

```
IntegerDigits[1/512, 2]
```

The digitsum of a positive integer is the sum of its digits. In the following plot, we show the digitsums associated with numbers between 0 and 1000. (We discuss the effect of the command `Apply` in Chapter 6; `ListPlot` and the related commands `PlotStyle`, `AxesLabel`, and `PointSize` are discussed in Chapter 1 of the Graphics volume [65✱].) (For some theoretical results on digitsums, see [2✱], [28✱], [29✱].)

```
ListPlot[Table[Apply[Plus, IntegerDigits[n]], {n, 0, 1000}],
         PlotStyle -> PointSize[0.005],
         AxesLabel -> (StyleForm[TraditionalForm[#]]& /@
                                          {n, digitsum[n]})]
```

Here, the number of ones in all binary representations of all numbers less than n − *mainAsymptoticTerm* are shown.

```
ListPlot[MapIndexed[# - #2[[1]]/2 Log[2, #2[[1]]]&,
                    Rest[FoldList[Plus, 0,
           Table[Count[IntegerDigits[n, 2], 1], {n, 2^12}]]]] // N,
         PlotStyle -> PointSize[0.002]]
```

The digit sums of the numbers n^k grow in average proportional to k with increasing k [38✱], [45✱]. The following graphic shows the digit sums divided by k for $n = 2, \ldots, 8$ as a function of k.

```
Show[Table[
  ListPlot[Table[Apply[Plus, IntegerDigits[n^k]]/k, {k, 1, 2500}],
           PlotStyle -> {PointSize[0.005], Hue[(n - 2)/8]},
           DisplayFunction -> Identity], {n, 2, 8}],
           DisplayFunction -> $DisplayFunction, PlotRange -> {0, 6},
           AxesLabel -> (StyleForm[TraditionalForm[#]]& /@
                                            {k, digitsum[n^k]/k})]
```

`IntegerDigits` can be used to find palindromic numbers in bases other than 10 [23✱]. The following function `palindromicBases` returns a list of sublists of the form `{`*base*`,` *digits*`}` for which a given integer *n* is palindromic.

```
palindromicBases[n_] :=
Module[{p}, Table[p = IntegerDigits[n, b];
                   If[p == Reverse[p], {b, p}, Sequence @@ {}],
                  {b, 2, n - 1}]]
```

The number 36960 is the smallest integer that is palindromic (and has at least two digits in each base) in 50 bases. Here are these 50 bases and the corresponding digits.

```
palindromicBases[36960]
```

Next, we demonstrate the decomposition of a few real numbers [31✱].

```
RealDigits[0.003476]

RealDigits[30003476.645]

RealDigits[0.125, 2]

RealDigits[0.2, 5]

RealDigits[3.0 7.34^2 + 5.0 7.34^1, 7.34]
```

The real and imaginary parts of complex numbers have to be decomposed separately.

```
RealDigits[2.34 + I 0.002345]

{RealDigits[2.34], RealDigits[0.002345]}
```

For rational numbers, `RealDigits` returns an exact answer.

`RealDigits[`*rationalNumber*`,` *base*`]`

produces a list characterizing the digits of the rational number *rationalNumber* in base *base* containing two elements. The first list contains the nonrepeating digits and a list of the repeating digits. The second element is the number of digits to the left of the decimal point. If *base* is not present, it is taken to be 10.

Here is a simple example.

```
RealDigits[12322/17]
```

We can compare the digits with a high-precision numerical approximation for 12322/17.

```
N[12322/17, 200]
```

To convert back from the result of `RealDigits` to a number, we can use the function `FromDigits`.

`FromDigits[`*nestedList*`,` *base*`]`

produces the real or rational number *x*, such that `RealDigits[`*x*`,` *base*`]=`*nestedList*`.`

Here, we convert back to the starting fraction 12 322 / 17.

```
FromDigits[%%]
```

`FromDigits` also works with symbolic input.

```
FromDigits[{{a1, a2, a3, a4, a5, {b1, b2, b3, b4, b5, b6}}, -2}]
```

Here is a plot of the length of the periodic part of the base *b* expansion of 1/12345.

```
ListPlot[Table[{b, Length[RealDigits[1/12345, b][[1, -1]]]},
               {b, 2, 1000}], PlotRange -> All]
```

If we just want to rewrite a given number in another base, we can use `BaseForm`.

`BaseForm[`*number*`,` *base*`]`

writes the number *number* in the base *base*. *base* must be an integer between 2 and 36.

```
BaseForm[512, 2]
```

For bases greater than 10, the numbers 10 through 36 are represented by the letters `a` through `z`.

```
BaseForm[32397578, 12]

BaseForm[
 10 36^36 + 11 36^35 + 12 36^34 + 13 36^33 + 14 36^32 + 15 36^31 +
 16 36^30 + 17 36^29 + 18 36^28 + 19 36^27 + 20 36^26 + 21 36^25 +
 22 36^24 + 23 36^23 + 24 36^22 + 25 36^21 + 26 36^20 + 27 36^19 +
 28 36^18 + 29 36^17 + 30 36^16 + 31 36^15 + 32 36^14 + 33 36^13 +
 34 36^12 + 35 36^11 +  9 36^10 +  8 36^09 +  7 36^08 +  6 36^07 +
  5 36^06  + 4 36^05  + 3 36^04 +  2 36^03 +  1 36^02, 36]
```

The second argument of `BaseForm` must be an integer between 2 and 36.

```
BaseForm[0.3, 0.3]
```

You can input integers in bases between 2 and 36 using the *base*^^*exponent* notation.

```
BaseForm[2621871, 23] // InputForm

23^^98b69
```

Be aware that `BaseForm` as a formatting function is limited to the use of alphanumeric characters. Using `RealDigits` or `IntegerDigits` allows the use of arbitrary bases.

If we are not interested in the single digits, but rather in the statistics of digits in a number, the function `DigitCount` comes in handy.

`DigitCount[`*integer*`,` *base*`]`

gives a list $\{s_1^{(base)}, s_2^{(base)}, \ldots, s_{base-1}^{(base)}, s_0^{(base)}\}$ of the number of digits $s_k^{(base)}$ of the integer *integer* in base *base*.

Here is a self-explanatory example.

```
DigitCount[122333444455555666666777777788888888999999999000000000, 10]
```

`DigitCount[`*integer*`, `*base*`, `*digit*`]`

gives $s_{digit}^{(base)}$, counting how often the digit *digit* occurs in the base *base* representation of the integer *integer*.

Here is a picture of the digit count of all digits of the number 100 in all bases $2 \le base \le 200$.

```
With[{n = 200},
Show[Graphics3D[{Thickness[0.002],
Table[{Hue[0.8 base/n],
       Line[Table[{base, digit, DigitCount[100, base, digit]},
                  {digit, 0, base - 1}]]}, {base, 2, n}]}],
                   Axes -> True, PlotRange -> {0, 2},
     BoxRatios -> {2, 1, 1}, ViewPoint -> {0, -3, 1},
      AxesLabel -> {"base", "digit", "n"}]]
```

At the end of this subsection, let us mention the two functions `IntegerPart` and `FractionalPart`.

`IntegerPart[`*realNumber*`]`

gives the integer part of the real number *realNumber*.

`FractionalPart[`*realNumber*`]`

gives the fractional part of the real number *realNumber*.

Here are some simple examples.

```
IntegerPart[11/2]

IntegerPart[-2.3]
```

Here the integer parts of $n \sin(n)$ for $1 \le n \le 10\,000$ are shown.

```
ListPlot[Table[IntegerPart[n Sin[n]], {n, 10^4}],
         PlotStyle -> {PointSize[0.002]},
         Frame -> True, Axes -> False];
```

Fractional parts are in most cases rewritten in the form *expr* − `IntegerPart[`*expr*`]`.

```
FractionalPart[Sin[3] + Exp[100]]

N[%, 100]
```

Here the fractional parts of $n \log(n)$ for $1 \le n \le 10\,000$ and of $10^9 / n$ for $20\,000 \le n \le 40\,000$ shown. The picture shows characteristic “empty spaces”.

```
Show[GraphicsArray[
ListPlot[#, PlotStyle -> {PointSize[0.002]}, Axes -> False,
            Frame -> True, DisplayFunction -> Identity]& /@
        N[{Table[FractionalPart[n Log[n]], {n, 10^4}],
           Table[{n, FractionalPart[10^9/n]}, {n, 20000, 40000}]}]]]
```

The next four pictures show the sums $f(n) = \sum_{k=0}^{n} (\text{frac}(k\,\alpha\,\pi) - 1/2)$ as a function of n. We use rational numbers near 1 for α and let n run up to 10^5. Depending on the “rationality” [69★], [52★], [37★] of α, we get curves that differ greatly in appearance.

```
fpsPlot[c_, n_] := With[{cn = N[c]},
ListPlot[FoldList[Plus, 0, Table[FractionalPart[k cn] - 1/2,
                  {k, n}]], PlotStyle -> PointSize[0.002],
         DisplayFunction -> Identity]]

(* four pictures *)
Show[GraphicsArray[
  {fpsPlot[(1 - 84 10^-7) Pi, 10^5], fpsPlot[(1 - 41 10^-7) Pi, 10^5]}]]

Show[GraphicsArray[
  {fpsPlot[(1 -  0 10^-7) Pi, 10^5], fpsPlot[(1 + 67 10^-7) Pi, 10^5]}]]
```

The fractional part function is a very useful construct for many iterative maps. The following is the Fibonacci chain map [48✱]. frac(x) denotes again the fractional part of x, sgn(x) the sign of x, and ϕ the golden ratio.

$$\{x_{n+1}, \varphi_{n+1}\} = \left\{-\frac{1}{x_n + \varepsilon + \alpha\,\mathrm{sgn}(\mathrm{frac}(n\,(\phi-1)) - (\phi-1))},\ \mathrm{frac}(\varphi_n + \phi - 1)\right\}$$

Being at the end of the first (nonintroductory) chapter, we will relax a moment and animate the Fibonacci chain map. For $\varepsilon = 1/2$, $x_0 = \pi$, $\varphi_0 = e$ we iterate the map 10000 times and display the resulting points $\{\tanh(x_k), \varphi_k\}$. We let α vary from 0.258 to 0.268. As visible from the graphics, the points collapse to curve segments. (At the current point, the reader should not analyze the following code; later we will use repeatedly similar constructions.)

```
f[α_, ε_, n_, {x0_, φ0_}] := FoldList[{-1/(#1[[1]] + ε -
              α Sign[FractionalPart[#2 (GoldenRatio - 1.)] -
                                         (GoldenRatio - 1.)]),
     FractionalPart[#1[[2]] + GoldenRatio - 1.]}&, {x0, φ0}, Range[n]]

fibanacciChainMapGraphics[α_, ε_, n_, {x0_, φ0_}] :=
Graphics[{PointSize[0.002],
  MapIndexed[{Hue[0.8 #2[[1]]/10^4], Point[#]}&,
             (* the scaled iterated values *)
             Tanh /@ Rest[f[α, 1/2, 10^4, N @ {Pi, E}]]]},
   Axes -> False, Frame -> True, PlotRange -> {{0, 1}, {0, 1}},
   FrameTicks -> None]

Show[GraphicsArray[fibanacciChainMapGraphics[#,
                    1/2, 10^4, N @ {Pi, E}]& /@ #]]& /@
                (* α-values *)
                Partition[Table[α, {α, 0.258, 0.268, 0.01/8}], 3]
```

```
Do[Show[fibanacciChainMapGraphics[α, 1/2, 10^4, N @ {Pi, E}]],
   {α, 0.258, 0.268, 0.01/100}];
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

Overview

We are at the end of the second chapter, and now it is time to give an overview of the functions discussed in this chapter. The overview will be computed using the function `ChapterOverview` from the package `` `ChapterOverview` ``. This package contains a list of all of the functions discussed in this book. The following information will be generated for each of the commands (in alphabetical order) found in every one of the sections:

■ The function.

■ Whether the function is an attribute. (We discuss the function `Attributes` in detail in Chapter 3; for the sake of uniformity, we use this form already here.)

■ Whether the function is an option. (Options are also treated in detail in Chapter 3.) If yes, the functions that have this option are listed.

■ The section in which the function was introduced. The appearance of "—" means that this function is not discussed in the *GuideBooks*.

The abbreviations P, G, N, and S stand for the Programming, Graphics, Numerics, and Symbolics volumes of the *GuideBooks*.

To simplify reading it, we assume that the file `ChapterOverview.m` is included in the directory containing packages (if it is placed in another directory, we would have to use the full path specification for the location of the file). We now load it. (The slightly complicated-looking input makes sure that the file `ChapterOverview.m` is found independent of the platform and independent of the location of the *GuideBooks* folder in the file system).

```
Get[ToFileName[ReplacePart[
          "FileName" /. NotebookInformation[EvaluationNotebook[]],
          "ChapterOverview.m", 2]]];
```

Here is a description of `ChapterOverview[`*subject*`,` *chapterNumber*`]`.

```
?ChapterOverview
```

Here is the overview of the functions discussed in this chapter.

```
ChapterOverview["Programming", 2]
```

Exercises

1.^{L1} What Is the Answer?

Predict the results of the following *Mathematica* inputs, and compare each prediction with the actual *Mathematica* output.

a) `b + a + a`

b) `2 + 4 + u + 8 + i + u - i`

c) `2 + 0I`

d) `Head[2 + 0I ]`

e) `0.0I - 0.0I`

f) `FullForm[0.0I - 0.0I ]`

g) `Infinity^Infinity`

h) `Infinity/Infinity`

i) `Infinity - Infinity`

j) `1/Indeterminate`

k) `FullForm[s + s^s/s - s]`

l) `Times[Times, Times]`

m) `Times[Times[], Times[]]`

n) `Times[Times[Times], Times[Times]]`

2.^{L1} `FullForm[`*expression*`]` with ()?

Try to find a *Mathematica* expression *expression* so that `FullForm[`*expression*`]` contains parentheses.

3.^{L1} `na38bvu94iwymmwpu1k5h6jhtye934` and `((1/2 + 1/4 I)^(7))^(1/7)`

a) What could the input be if the output is `na38bvu94iwymmwpu1k5h6jhtye934`. Give at least two possible answers.

b) Why is the result of inputting `((1/2 + 1/5 I)^(7))^(1/7)` just `1/2 + 1/5 I`, but the result of `((1/2 + 1/4 I)^(7))^(1/7)` is `(-139/8192 - 29 I/16384)^(1/7)` and not `1/2 + 1/4 I`?

c) Find a built-in function *f*, such that the input `Head @ (Im[`*f*`[3]] // N)` returns the output `Complex`.

4.^{L2} `Level`, `Depth`, and `Part`

Analyze the following expression as a *Mathematica* expression:

$$expr = \sin\Big(\tan(1 + e^{-x}) + x^x - \ln(\ln(r\,t + a\,x)) + d(x) + x(x)\arccos\big(\arcsin\big(x^2\big)\big) + h(h(h(i)))\Big)$$

What is its depth? Examine all possible levels. Where does x appear? Investigate all sensible values of `Part[`*expr*`,` *nonNegativeNumber*`]`.

5.^{L2} `Level[`*expr*`, {-2, 2}]` versus `Level[`*expr*`, {2, -2}]`

What are the results of the following two inputs?

```
Level[Sin[3 x + Cos[6/(t + Tan[r])]/Exp[-x^2]], {-2, 2}]

Level[Sin[3 x + Cos[6/(t + Tan[r])]/Exp[-x^2]], {2, -2}]
```

6.^{L2} Branch Cuts

a) Discuss the location of the branch cuts of the function $f(z) = 1\Big/\left(z^4\right)^{\frac{1}{4}}$ in *Mathematica* (meaning `1/(z^4)^(1/4)`). What are the values of the function $f(z)$ on the other sheets of the Riemann surface?

b) Theoretically, the location of branch cuts of an analytic function is not fixed. But by using the built-in functions of *Mathematica*, the branch cuts of nested functions built from the built-in functions are determined. Determine the location in *Mathematica* of the branch cuts of the function $f(z) = \sqrt{z+1/z}\ \sqrt{z-1/z}$.

c) Determine the branch points and the branch cuts of the following function $w(z)$: $w(z) = \arctan(\tan(z/2)/2)$.

d) Characterize the function `Sqrt[z] - 1/Sqrt[1/z]`.

e) Characterize the function `1/(z + Sqrt[z^2])`.

f) Discuss the branch cut and branch point structure of the function $g(z) = \sqrt{z + \sqrt{z-1}\ \sqrt{z+1}}$ in *Mathematica*.

g) Describe the branch cut location of the function $f(z)$ =`1/Log[Exp[1/z]]`.

h) Discuss the branch point and branch cut structure of the functions arccoth, arccosh, and arcsech. In *Mathematica* they are defined as

```
ArcCoth[z] = Log[1 + 1/z]/2 - Log[1 - 1/z]/2
ArcCosh[z] = Log[z + Sqrt[z + 1] Sqrt[z - 1]]
ArcSech[z] = Log[Sqrt[1/z + 1] Sqrt[1/z - 1] + 1/z].
```

How many different sheets does one reach by encircling the origin and ± 1 (on the corresponding Riemann surface) at various radii? What happens if one moves around the eight-shaped contour $\{2\cos(\varphi), \sin(\varphi)\}$? Is infinity a branch point? What are the differences of the function values across the branch cuts?

7.^{L2} "Strange" Analytic Functions

For all parts of this exercise, use only analytic functions like `Exp`, `Log`, `Power`, `Sqrt`, … as building blocks, do not use functions like `Abs`, `Re`, ….

a) Construct a function f that is 1 on the unit circle $|z| = 1$ and 0 everywhere else (with the possible exception of a finite number of other points).

b) Construct a function f that is 1 inside the unit circle and 0 everywhere else.

c) Construct a function f that evaluates to 1 at $x = 0$ and to 0 at every other real x.

d) Construct a function f that evaluates 1 in the open interval (0, 1) and to 0 at every other real x.

e) Construct a function f that is equal to the staircase function $\lfloor x \rfloor$ for real values x (with the possible exception of the points of discontinuity of $\lfloor x \rfloor$). (Here, $\lfloor x \rfloor$ is equal to the smallest integer less than or equal to x.)

f) Construct a function f that is equal to the "castle rim function" $x \bmod 2$ (with the possible exception of the points of discontinuity of $x \bmod 2$).

g) Construct a function f that is equal to the sawtooth function $1 - 2\,|[x] - x/2|$, where $[x]$ denotes the rounding to nearest integer to x.

8.^{L2} `ArcTan[(x + 1)/y] - ArcTan[(x - 1)/y]` Picture

Predict how a picture of $\tan^{-1}((x+1)/y) - \tan^{-1}((x-1)/y)$ over the real x,y-plane will look. We get such a picture in *Mathematica* by using the following code.

```
f[x_,y_] := ArcTan[(x + 1)/y]- ArcTan[(x - 1)/y]
ε = 10^-14;
Plot3D[Evaluate[f[x,y]], {x, -Pi, Pi}, {y, ε, Pi},
       PlotPoints -> 50, PlotRange -> All];
```

9.[L2] `ArcSin[ArcSin[z]]` Picture

Predict how a 3D picture of $\mathrm{Im}\left(\sin^{-1}\left(\sin^{-1}(x+i\,y)\right)\right)$ over the real x,y-plane will look. We get such a picture in *Mathematica* by using the following code.

```
Plot3D[Im[ArcSin[ArcSin[x + I y]]], {x, -3, 3}, {y, -3, 3}, PlotPoints -> 40];
```

10.[L2] Singularities of tanh(sinh(cot(*z*))), $\exp\left(\ln^{i\pi}(z)\right)$ Properties

a) At which points z does the function $w(z) = \tanh(\sinh(\cot(z)))$ have singularities? What kind of singularities?

b) Describe the branch cuts of the function $f(z) = \arg\left(\exp\left(\ln^{i\pi}(z)\right)\right)$ over the complex z-plane. Express $\arg(f(z))$ in an explicit real way as a function of $|z|$ and $\arg(z)$ and give a qualitative description of $\arg(f(z))$ over the complex z-plane.

11.[L1] `Exp[-1/Im[1/(-Log[Infinity] + 2)^2]]`

Predict the result of evaluating `Exp[-1/Im[1/(-Log[Infinity] + 2)^2]]`.

12.[L1] Predict the Result

Predict the results of

```
N[(1 - 10^-21) Exp[I 2], 22]^Infinity
```

and

```
N[(1 - 10^-23) Exp[I 2], 22]^Infinity.
```

13.[L1] $\tan(k/\alpha) + \tan(\alpha\, k)$ Picture

The following input defines a function `tanPicture` that displays the set of points $\{k, \tan(k\,\alpha) + \tan(k/\alpha)\}$ for $k = 1, \ldots, 20\,000$. Find different real values of α such that `tanPicture[α]` looks "qualitatively different".

```
tanPicture[α_] :=
ListPlot[Table[Tan[α k] + Tan[1/α k], {k, 20000}],
         PlotStyle -> {PointSize[0.001]}, PlotRange -> {-2, 2},
         Frame -> True, Axes -> False, FrameTicks -> None]
```

Solutions

1. What Is the Answer?

We let the inputs run, and comment only on possible problems and things that might not be obvious.

a) `a + a` is simplified to `2a`, and the expression is reordered.

```
b + a + a
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

b) Simplifying and reordering gives the following expression.

```
2 + 4 + u + 8 + i + u - i
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

c) `0*I` is identically 0.

```
2 + 0 I
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

d) It is an integer.

```
Head[%]
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

e) The two `(0.01 I)`s are treated as distinct approximate numbers with vanishing real parts. They could come from distinct calculations, and they may differ for digits beyond the machine precision. Thus, the result is `0.0 I`.

```
0.0 I - 0.0 I
```

On the other hand, here is another example with two exact numbers. They cancel to 0.

```
Complex[0, 0] - Complex[0, 0]
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

f) The `FullForm` of the complex number `0 + 0.0I` is given.

```
FullForm[0.0 I - 0.0 I]
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

g) The result is ∞ in any case in magnitude, but because we do not know the direction e.g., $\lim_{x\to\infty}(x+i/x)^{\exp(x)}$ for large real x, is actually `ComplexInfinity` [35✱].

```
Infinity^Infinity
```

Similarly, `1^Infinity` and `Infinity^0` also evaluate to `Indeterminate`.

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

h) Depending on the nature of the two infinity results, any result is possible. Therefore, we get an indefinite result. (The three expressions x/x^2, x^2/x, x/x yield different limit values.)

```
Infinity/Infinity
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

i) The difference is unknown in magnitude, so `Indeterminate` is returned.

```
Infinity - Infinity
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

j) The result of an arithmetic operation with something indeterminate remains indeterminate.

```
1/Indeterminate
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

k) The first and last `s` cancel out so that `s^s/s` remains. `(s^s)/s` results from this. Now, the `s` in the denominator is canceled, giving `s^(s - 1)`, and after an alphabetical reordering, we get `s^(-1 + s)`.

```
FullForm[s + s^s/s - s]
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

l) `Times[Times, Times]` is precisely the product (because of head `Times`) of the two symbols `Times` and `Times`.

```
Times[Times, Times]
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

m) Because `Times` is called with no arguments, it is equal to 1.

```
Times[]
```

Because $1 \times 1 = 1$, it follows that we get this result.

```
Times[Times[], Times[]]
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

n) Because `Times` is called with one argument, the result is the argument itself.

```
Times[Times]
```

We get `Times`^{2}.

```
Times[Times[Times], Times[Times]]
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

2. `FullForm[`*expression*`]` with ()?

No `FullForm[`*something*`]` with parentheses exists if *something* is a string-free *Mathematica* expression. The order of the evaluation/structure is uniquely determined by the brackets `[]`. Of course, *something* could contain a string with parentheses.

3. `na38bvu94iwymmwpu1k5h6jhtye934` and `((1/2 + 1/4 I)^(7))^(1/7)`

a) One obvious solution would be just to use a variable containing the displayed sequence. Another solution is provided.

```
BaseForm[23 36^29 + 10 36^28 +  3 36^27 +  8 36^26 +
         11 36^25 + 31 36^24 + 30 36^23 +  9 36^22 +
          4 36^21 + 18 36^20 + 32 36^19 + 34 36^18 +
         22 36^17 + 22 36^16 + 32 36^15 + 25 36^14 +
         30 36^13 +  1 36^12 + 20 36^11 +  5 36^10 +
         17 36^09 +  6 36^08 + 19 36^07 + 17 36^06 +
         29 36^05 + 34 36^04 + 14 36^03 +  9 36^02 +
          3 36^1 +   4 36^00, 36]
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

b) Let us first confirm the claims made in the statement of the exercise.

```
((1/2 + 1/5 I)^(7))^(1/7)
```

```
((1/2 + 1/4 I)^(7))^(1/7)
```

In fact, the second output was not evaluated to `1/2 + 1/4 I`. The reason is not a bug in *Mathematica* or weakness; rather it is the different argument of the complex number exponentiated. After taking the seventh power, these are the arguments of the resulting quantities.

```
7 Arg[N[1/2 + 1/5 I]]
```

```
7 Arg[N[1/2 + 1/4 I]]
```

In the second case, the result is larger than π. But the argument convention in *Mathematica* is that the argument of every complex number lies in the range $-\pi < arg \leq \pi$. So this argument must be reduced modulo π. The resulting number has a negative argument.

```
(1/2 + 1/4 I)^(7)
```

```
N[Arg[%]]
```

Now, taking this number to the power $1/7$ means taking the seventh root of the absolute value and the seventh part of the argument, which does not give `1/2 + 1/4 I`, but rather gives a seventh root that is not further automatically simplified.

```
((1/2 + 1/4 I)^(7))^(1/7)
```

This quantity is clearly distinct from `1/2 + 1/4 I`.

```
N[% - (1/2 + 1/4 I)]
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

c) To return the head `Complex`, we need a numericalized expression that is complex (potentially with a vanishing imaginary part). Because `Im[`*x*`]` will return a real number for an approximate number *x*, `Im[f[3]]` must autoevaluate to an expression not having the head `Im`. This is, for instance, the case for *f* = `ArcCos`.

```
Im[ArcCos[3]]
```

Numericalizing the last expression means to numericalize the two factors *i* and arccos(3). The result is an approximate number with an (approximately) vanishing imaginary part.

```
N[%]
```

So, the function *f* = `ArcCos` yields the head `Complex` for the original input.

```
f = ArcCos;
Head @ (Im[f[3]] // N)
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

4. Level, Depth, and Part

Here is the expression.

```
big = Sin[Tan[1 + Exp[-x]] + x^x - Log[Log[r t + a x]] +
          d[x] + x[x] - ArcCos[ArcSin[x^2]] + h[h[h[i]]]]
```

Its depth is 8.

```
Depth[big]
```

Here are its positive levels. First is the expression itself.

```
Level[big, {0}]
```

Here is the level 1.

```
Level[big, {1}]
```

Here is the level 2.

```
Level[big, {2}]
```

Here is the level 3.

```
Level[big, {3}]
```

Here is the level 4.

```
Level[big, {4}]
```

Here is the level 5.

```
Level[big, {5}]
```

Here is the level 6.

```
Level[big, {6}]
```

And here is the level 7.

```
Level[big, {7}]
```

The level 8 does not exist. The depth is equal to the number of levels + 1.

```
Level[big, {8}]
```

Here is an analysis of the expression from its roots.

```
Level[big, {-1}]

Level[big, {-2}]

Level[big, {-3}]

Level[big, {-4}]

Level[big, {-5}]

Level[big, {-6}]

Level[big, {-7}]

Level[big, {-8}]
```

As with level 8, no level −9 exists for `big`.

```
Level[big, {-9}]
```

Now, we consider `x`.

```
Position[big, x]

Length[%]
```

`x` appears exactly eight times. Here are these eight positions.

```
big[[1, 1, 1]]

big[[1, 1, 2]]

big[[1, 2, 2, 1, 1, 1]]

big[[1, 3, 1]]

big[[1, 5, 2, 1, 1, 2, 2]]

big[[1, 6, 1, 2, 2, 2]]

big[[1, 7, 0]]

big[[1, 7, 1]]
```

The expression consists of exactly 60 parts (not including itself), each of which can be obtained using `Part`.

```
allParts = Level[big, {1, Infinity}, Heads -> True]

Length[allParts]
```

Of the 60 parts, 40 are distinct. (The function `Union` is discussed in Chapter 6; it eliminates duplicate elements.)

```
Length[Union[allParts]]
```

Here are all 60 parts. To save space, we let *Mathematica* determine the positions of the individual components. The way the program works will become clear in the course of studying this book; here, we are only interested in the result.

```
MapIndexed[(* the subexpression *)
  (CellPrint[Cell[TextData[{"∘ Part number ",
    ToString[#2[[1]]], " is ",
      StyleBox[ToString[#, InputForm], "MR"],
   (* where does this subexpression occur? *)
   " and occurs at the following positions: ",
              StyleBox[ToString[Position[big, #, Heads -> True],
                                  InputForm], "MR"]}], "PrintText"]])&,
              Take[allParts, 3], {1}];
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

5. `Level[`*expr*`, {-2, 2}]` versus `Level[`*expr*`, {2, -2}]`

Here is the expression under consideration.

```
expr = Sin[3 x + Cos[6/(t + Tan[r])]/Exp[-x^2]]
```

As discussed, `Level[`*expr*`, {`n_1`, `n_2`}]` gives all parts of *expr* that are at level n_1 or below and that are at the same time at level n_2 or above. These are all the nonempty positive and negative levels.

```
(* ∘ stands again for Mathematica generated text *)
Do[CellPrint[Cell[TextData[{"∘ Elements of ", StyleBox["expr", "MR"],
                              " at level level "<> ToString[i] <>":"}],
              "PrintText"]]; Print[Level[expr, {i}]],
   {i, 0, 8}]
```

Now, we start from the roots.

```
(* ∘ stands again for Mathematica generated text *)
Do[CellPrint[Cell[TextData[{"∘ Elements of ", StyleBox["expr", "MR"],
                              " at level level "<> ToString[i] <>":"}],
              "PrintText"]]; Print[Level[expr, {i}]],
  {i, 0, -9, -1}]
```

`Level[expr, 2, -2]` is the intersection of all levels between the positive levels 2 and 8 and all negative levels between −8 and −2.

```
Level[expr, {2, -2}]
```

In Chapter 6, we will discuss the following construction, which explicitly determines this intersection of the levels needed here. (The order of the elements is different from the last output.)

```
Intersection[Flatten @ Table[Level[expr, {i}], {i, 2, 8}],
             Flatten @ Table[Level[expr, {i}], {i, -2, -8, -1}] ]
```

`Level[expr, -2, 2]` is the intersection of all levels between the positive levels 0 and 2 and all negative levels between −2 and −1.

```
Level[expr, {-2, 2}]
```

Here again, this intersection (we will discuss the function `Intersection` in Chapter 6) is determined explicitly.

```
Intersection[Flatten @ Table[Level[expr, {i}], {i, -1, -2, -1}],
             Flatten @ Table[Level[expr, {i}], {i, 0, 2}] ] // Union
```

For most expressions, `Level[`*expression*`, {-`*i*`, `*i*`}]` is different from `Level[`*expression*`, {`*i*`, -`*i*`}]`.

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

6. Branch Cuts

a) The power function has a branch cut along the negative real axis, which means that $1\big/\left(z^4\right)^{1/4}$ has branch cuts when z^4 is a negative real number. Using the representation $z = e^{i\varphi}$, we get the following four possibilities: $e^{4i\varphi} = e^{i\pi}$, $e^{4i\varphi} = e^{3i\pi}$, $e^{4i\varphi} = e^{5i\pi}$, $e^{4i\varphi} = e^{7i\pi}$. These relations mean that $1\big/\left(z^4\right)^{1/4}$ has branch cuts along the rays $z = r\,e^{i\pi/4}$, $z = r\,e^{3i\pi/4}$, $z = r\,e^{5i\pi/4}$, and $z = r\,e^{7i\pi/4}$. The function values of $f(z)$ on one sheet of the Riemann surface of $1\big/\left(z^4\right)^{1/4}$ are immediately given, and the function values on the other three sheets are obtained by letting φ in $z = e^{i\varphi}$ vary over the range $(0, 8\pi)$, which means over four copies of the original z-plane. So, the other three function values are given by $e^{i\pi/2}\,f(z)$, $e^{i\pi}\,f(z)$, and $e^{3i\pi/2}\,f(z)$.

This graphic shows the imaginary part of the four sheets.

```
Show[GraphicsArray[Table[Show[Table[
   ParametricPlot3D[{r Cos[φ], r Sin[φ],
                     Im[1/(Exp[2Pi i I/4] ((r Exp[I φ])^4)^(1/4))],
                     {(* no individual polygon edges *) EdgeForm[]}},
                    {r, 1/2, 2}, {φ, φ0 + 10^-8, φ0 - 10^-8 + Pi/2},
                    DisplayFunction -> Identity],
                 {φ0, Pi/4, 2Pi - Pi/4, Pi/2}],
            PlotRange -> {{-2, 2}, {-2, 2}, {-2, 2}}],
      {i, 0, 3}], GraphicsSpacing -> 0]]
```

The real part looks similar.

```
Show[GraphicsArray[Table[Show[Table[
   ParametricPlot3D[{r Cos[φ], r Sin[φ],
                     Re[1/(Exp[2Pi i I/4] ((r Exp[I φ])^4)^(1/4))],
                     {(* no individual polygon edges *) EdgeForm[]}},
                    {r, 1/2, 2}, {φ, φ0 + 10^-8, φ0 - 10^-8 + Pi/2},
                    DisplayFunction -> Identity],
                 {φ0, Pi/4, 2Pi - Pi/4, Pi/2}],
           PlotRange -> {{-2, 2}, {-2, 2}, {-2, 2}}],
       {i, 0, 3}], GraphicsSpacing -> 0]]
```

Combining all sheets from the last four pictures in one picture, we get the complete Riemann surface of $1\big/\left(z^4\right)^{1/4}$. The four sheets are not connected [8✱], [51✱].

```
Show[%[[1]], DisplayFunction -> $DisplayFunction]
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

b) As a first orientation, take a look at a 3D graphic of the imaginary part of $f(z)$.

```
Plot3D[Im[Sqrt[(x + I y) + 1/(x + I y)] Sqrt[(x + I y) - 1/(x + I y)]],
       {x, -2, 2}, {y, -2, 2}, PlotPoints -> 50]
```

This picture indicates a branch cut along the left half of the unit circle and along the real line between -1 and 1. Be aware that for a generic complex z we have

$$\sqrt{z+\frac{1}{z}}\;\sqrt{z-\frac{1}{z}} \neq \sqrt{\left(z+\frac{1}{z}\right)\left(z-\frac{1}{z}\right)}.$$

Again, a picture shows this clearly.

```
Plot3D[Im[Sqrt[((x + I y) + 1/(x + I y)) ((x + I y) - 1/(x + I y))]],
       {x, -2, 2}, {y, -2, 2}, PlotPoints -> 50]
```

Now let us analytically tackle the problem of the locations of the branch cuts of $f(z)$. The `Sqrt` function in *Mathematica* has a branch cut for negative arguments, which means that the branch cuts of $f(z)$ are determined by the following parametric representation (in dependence of *negativeRealNumber*):

$$z \pm \frac{1}{z} = negativeRealNumber.$$

Solving the first of these two equations gives

$$z_{1,2} = \frac{negativeRealNumber}{2} \mp \sqrt{\left(\frac{negativeRealNumber}{2}\right)^2 - 1}\;.$$

For $-2 \leq negativeRealNumber \leq 0$, we have for z_1

$$\begin{aligned}\mathrm{Re}(z_1) &= \frac{negativeRealNumber}{2}\\ \mathrm{Im}(z_1) &= -\sqrt{1-\left(\frac{negativeRealNumber}{2}\right)^2}\;.\end{aligned}$$

These formulas describe the part of the unit circle in the third quadrant. For $\leq negativeRealNumber \leq -2$, we have for z_1

$$\begin{aligned}\mathrm{Re}(z_1) &= \frac{negativeRealNumber}{2} - \sqrt{\left(\frac{negativeRealNumber}{2}\right)^2 - 1}\\ \mathrm{Im}(z_1) &= 0.\end{aligned}$$

These formulas describe all points on the real line that are to the left of -1. A similar analysis for z_2 shows that for $-2 \leq negativeRealNumber \leq 0$, the part of the unit circle in the second quadrant is covered and $\leq negativeRealNumber \leq -2$, which is the interval $(-1, 0)$ of the real line. Again, a visualization confirms the so-located branch cuts.

```
Plot3D[Im[Sqrt[((x + I y) + 1/(x + I y))]], {x, -2, 2}, {y, -2, 2},
       PlotPoints -> 50]
```

The second square root $(z - 1/z)^{1/2}$ gives the following two parametric representations for possible branch cuts of the function under consideration here.

$$z_{1,2} = \frac{negativeRealNumber}{2} \mp \sqrt{\left(\frac{negativeRealNumber}{2}\right)^2 + 1}\;.$$

This plot shows immediately that the branch cut of this part is $(-\infty, 1)$, as it is also shown in the following picture.

```
Plot3D[Im[Sqrt[((x + I y) - 1/(x + I y))]], {x, -2, 2}, {y, -2, 2},
       PlotPoints -> 50]
```

Now, we have all possible branch cut locations collected. Along $(-\infty, 1)$, the branch cuts of $(z+1/z)^{1/2}$ and $(z-1/z)^{1/2}$ coincide. As a result, the corresponding jumps may compensate each other or may add to each other. To determine when which situation happens, we look at the value of the two arguments of the square roots for $x<0$, ε small, which means just below and above the potential branch cut.

$$x+i\,\varepsilon \pm \frac{1}{x+i\,\varepsilon} \;= x \pm \frac{1}{x} + i\,\varepsilon\left(1 \mp \frac{1}{x^2}\right) + O(\varepsilon^2).$$

This process shows that for $x<-1$, the imaginary parts of $x+i\,\varepsilon+1/(x+i\,\varepsilon)$ and $x+i\,\varepsilon-1/(x+i\,\varepsilon)$ have the same sign, and the discontinuities in the product of the two square roots just cancel. So, no branch cut exists in the interval $(-\infty, -1)$. For $-1<x<0$, the imaginary parts of $x+i\,\varepsilon+1/(x+i\,\varepsilon)$ and $x+i\,\varepsilon-1/(x+i\,\varepsilon)$ have opposite signs, and as a result, in this interval, a branch cut occurs.

To end this discussion, let us have a more detailed look at $f(z)$. We see the branch cuts more clearly, when the steep vertical walls are not shown. (Chapter 2 of the Graphics volume [65✱] discusses in detail how to make graphics similar to the next two.)

```
ε = 10^-6;
Show[Apply[ParametricPlot3D[
{r Cos[φ], r Sin[φ], Im[Sqrt[r Exp[I φ] + 1/(r Exp[I φ])]*
                        Sqrt[r Exp[I φ] - 1/(r Exp[I φ])]],
 (* thin polygon edges *) EdgeForm[Thickness[0.001]]}, ##,
 DisplayFunction -> Identity]&,
    (* all parts divided by branch cuts *)
 {{{r, ε, 1 - ε}, {φ, ε, Pi - ε}, PlotPoints -> {12, 30}},
  {{r, ε, 1 - ε}, {φ, Pi + ε, 2Pi - ε}, PlotPoints -> {12, 30}},
  {{r, 1 + ε, 2}, {φ, 0, 2Pi}, PlotPoints -> {12, 59}}}, {1}],
     DisplayFunction -> $DisplayFunction, PlotRange -> {-3, 3}]
```

Because $f(z)$ has a branch, the last picture shows just one of two sheets of the Riemann surface of $f(z)$. Because of the `Sqrt` in the function under consideration, it is easy to get the second sheet. Here, the whole Riemann surface is shown.

```
Show[{%, (* the other sheet *)
Show[Apply[ParametricPlot3D[
{r Cos[φ], r Sin[φ]Sin[φ], Im[-Sqrt[r Exp[I φ] + 1/(r Exp[I φ])]*
 Sqrt[r Exp[I φ] - 1/(r Exp[I φ])]], EdgeForm[Thickness[0.001]]}, ##,
 DisplayFunction -> Identity]&,
   (* all parts divided by branch cuts *)
 {{{r, ε, 1 - ε}, {φ, ε, Pi - ε}, PlotPoints -> {12, 30}},
  {{r, ε, 1 - ε}, {φ, Pi + ε, 2Pi - ε}, PlotPoints -> {12, 30}},
  {{r, 1 + ε, 2}, {φ, 0, 2Pi}, PlotPoints -> {12, 59}}}, {1}],
     DisplayFunction -> Identity,
     PlotRange -> {-3, 3}]}, Axes -> False, Boxed -> False,
     ViewPoint -> {1.55, -1.4, 1.5},
     DisplayFunction -> $DisplayFunction]
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

c) Let us first have a look at the function under consideration.

```
Plot3D[Re[ArcTan[Tan[(x + I y)/2]/2]], {x, 0, 6Pi}, {y, -3, 3},
       PlotPoints -> 30]
```

We see a couple of branch cuts parallel to the imaginary axis. The function `ArcTan` has two branch points at i and $-i$, and the complex plane is cut along $(i, i\,\infty)$ and $(-i, -i\,\infty)$. Solving $\tan(z/2)/2 = \pm i$ for z, we get the following for the location of the branch points of $\arctan(\tan(z/2)/2)$:

$$z = \pm 2 \arctan(2\,i)$$
$$z_k = \pm 2 \ln\left(\sqrt{3}\right) i + (2\,k+1)\,\pi\,,\ k \in \mathbb{Z}.$$

The second formula follows after simplification and takes the periodicity of tan into account. In *Mathematica*, we can get the this simplification by using `ComplexExpand`.

```
2 ArcTan[2I] // ComplexExpand

Tan[I Log[Sqrt[3]] + Pi/2]/2

Tan[-I Log[Sqrt[3]] + Pi/2]/2
```

Now, let us determine the location of the branch cuts. $\tan(i\,t + \pi/2)$ is purely imaginary for real t.

```
Tan[I t/2 + Pi/2]/2
```

The absolute value of $i/2 \coth(t/2)$ is greater than 1 in the range $-2 \ln \sqrt{3} < t < 2 \ln \sqrt{3}$.

```
Plot[Im[Tan[I t/2 + Pi/2]/2], {t, -2Log[Sqrt[3]], 2Log[Sqrt[3]]},
     PlotRange -> {-4, 4}]
```

From these observations, it follows that the branch cuts of $\arctan(\tan(z/2)/2)$ are the intervals $\left[-2 \ln \sqrt{3}\; i + (2\,k+1)\,\pi, 2 \ln \sqrt{3}\; i + (2\,k+1)\,\pi\right]$, $k \in \mathbb{N}$.

By excluding the branch cuts from the x,y-region covered in the above picture, we can make a more appropriate picture of the function $\arctan(\tan(z/2)/2)$. Here is the definition for one sheet of the Riemann surface of this function. δ translates the picture vertically.

```
sheet[δ_] :=
Block[{$DisplayFunction = Identity, ε = 10^-10},
{ (* 0 ≤ Re(z) < π *)
 Plot3D[Re[ArcTan[Tan[(x + I y)/2]/2] + δ],
        {x, 0, Pi - ε}, {y, -6 Log[Sqrt[3]], 6 Log[Sqrt[3]]},
        PlotPoints -> {15, 31}],
 (* π < Re(z) < 3π *)
 Plot3D[Re[ArcTan[Tan[(x + I y)/2]/2] + δ],
        {x, Pi + ε, 3Pi - ε}, {y, -6 Log[Sqrt[3]], 6 Log[Sqrt[3]]},
        PlotPoints -> {30, 31}],
 (* 3π < Re(z) < 5π *)
 Plot3D[Re[ArcTan[Tan[(x + I y)/2]/2] + δ],
        {x, 3Pi + ε, 5Pi - ε}, {y, -6 Log[Sqrt[3]], 6 Log[Sqrt[3]]},
        PlotPoints -> {30, 31}],
 (* 5π < Re(z) ≤ π *)
 Plot3D[Re[ArcTan[Tan[(x + I y)/2]/2] + δ],
        {x, 5Pi + ε, 6Pi}, {y, -6 Log[Sqrt[3]], 6 Log[Sqrt[3]]},
        PlotPoints -> {15, 31}]}]
```

This picture shows the principal sheet of $\arctan(\tan(z/2)/2)$.

```
Show[sheet[0], BoxRatios -> {2, 1, 1/2}];
```

Taking into account the $(2\,k+1)\,\pi$ term of z_k, we can display some sheets of the Riemann surface under consideration.

```
Show[{sheet[0], sheet[Pi/2], sheet[-Pi/2]},
     ViewPoint -> {1, -2.4, 1.5}, BoxRatios -> Automatic,
     AxesLabel -> {x, I y, None}]
```

Here, only one half of the last picture is shown to provide a better view of the connections between the sheets.

```
Show[%, PlotRange -> {All, {0, Pi}, All}, ViewPoint -> {-3, -2, 1}];
```

For a discussion of a general method to determine branch cuts of functions built from functions with known branch cuts, see [22✱].

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

d) Here, the function under consideration is defined.

```
f[z_] := Sqrt[z] - 1/Sqrt[1/z]
```

At a first view, we may think that the function is identically zero. A plot also suggests this. (We multiply the function values by $10^{machinePrecision}$.)

```
Plot3D[10^$MachinePrecision Abs[f[x + I y]], {x, -2, 2}, {y, -2, 2}];
```

But *Mathematica* does not automatically simplify this function to zero.

```
f[z]
```

And this absence of “simplification” is not the case because of the single point $z = 0$. Indeed, `f[z]` is not zero everywhere in the complex z-plane (and, of course, undefined at $z = 0$).

```
f[-2]
```

Now let us determine where `f[z]` does not vanish. The operation $z \longrightarrow 1/z$ maps the whole complex plane onto the whole complex plane. The lower half-plane is mapped onto the upper half-plane and vice versa. The `Sqrt` function has a branch cut along the negative real axis with continuity from above, which means that `f[z]` vanishes everywhere except along the negative real axis where the two terms `Sqrt[z]` and `Sqrt[1/z]` do not cancel but are the same.

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

e) Here is the function defined.

```
f[z_] := 1/(z + Sqrt[z^2])
```

A first view shows that the function is finite in the right half-plane. (We turn off some messages.)

```
Off[Power::infy]; Off[Plot3D::plnc]; Off[Plot3D::gval];
Plot3D[Abs[f[x + I y]], {x, -2, 2}, {y, -2, 2},
       PlotRange -> {-5, 5}, ClipFill -> None, PlotPoints -> 20]
```

In the left half-plane, the function is `ComplexInfinity`. (This is the reason for the turned off error messages in the last input.) For the right half-plane, *Mathematica* can simplify the function `f`.

```
Simplify[f[z], Re[z] > 0]
```

It remains to investigate the behavior of `f` on the imaginary axis. A sample input shows that $f(z)$ is finite on the positive imaginary axis.

```
f[2 I]
```

```
f[-2 I]
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

f) We start by investigating the function under the square root.

```
f[z_] := z + Sqrt[z - 1] Sqrt[z + 1];
```

Here is a graphic of its real and imaginary parts.

```
Show[GraphicsArray[{
          (* show real and imaginary parts *)
          Plot3D[Evaluate[Re[f[x + I y]]], {x, -2, 2}, {y, -2, 2},
                 PlotPoints -> 40, DisplayFunction -> Identity],
          Plot3D[Evaluate[Im[f[x + I y]]], {x, -2, 2}, {y, -2, 2},
                 PlotPoints -> 40, DisplayFunction -> Identity]}]]
```

$z = \pm 1$ are branch points coming from $\sqrt{z-1}\ \sqrt{z+1}$. The branch cut connecting them is clearly visible. A graphics of the absolute value of $f(z)$ shows that nowhere we have $f(z) = 0$.

```
Plot3D[Evaluate[Abs[f[x + I y]]],
       {x, -2, 2}, {y, -2, 2}, PlotPoints -> 40]
```

Along the real axis, we have the following behavior: For $|x| > 1$, the function is purely real, and for $x < 1$, the function $f(x)$ is negative.

```
Plot[Evaluate[{Re[f[x]], Im[f[x]]}], {x, -3, 3},
     PlotStyle -> {Hue[0], Hue[0.74]},
     Frame -> True, Axes -> False]
```

Now, let us look at the function $g(z)$.

```
g[z_] := Sqrt[z + Sqrt[z - 1] Sqrt[z + 1]]
```

In addition to the two branch points ± 1 and the branch cut joining them, we now see a branch cut to the left of $z = -1$ along the negative real axis.

```
Show[GraphicsArray[{
          (* show real and imaginary parts *)
          Plot3D[Evaluate[Re[g[x + I y]]], {x, -2, 2}, {y, -2, 2},
                 PlotPoints -> 40, DisplayFunction -> Identity],
          Plot3D[Evaluate[Im[g[x + I y]]], {x, -2, 2}, {y, -2, 2},
                 PlotPoints -> 40, DisplayFunction -> Identity]}]]
```

The branch cut along the negative imaginary axis is not related to the branch point $z = 1$ from the inner square root. Interestingly, at $z = -1$, one immediately "jumps" onto the branch cut of the outer square root function without ever passing the "corresponding branch point $z = 0$". The branch cut of the square root function extends from -∞ to 0. The argument of square root assumes the value -∞ at $z = -\infty$ of the first sheet of $z + \sqrt{z-1}\ \sqrt{z+1}$ (this is the sheet chosen by *Mathematica*) and the value 0 at $z = -\infty$ of the other sheet $z - \sqrt{z-1}\ \sqrt{z+1}$. So the branch cut visible in the picture runs in a loop-like form from $-\infty$ to -1 and then back to -∞.

We can get a better impression about this function by looking at all its four sheets. The other sheets of the two square root functions are easily obtained as $\pm\sqrt{\ldots}$.

```
sheetg[j_, k_, z_] := (-1)^j Sqrt[z + (-1)^k Sqrt[z - 1] Sqrt[z + 1]];
```

Here the four sheets in the neighborhood of the two branch points ± 1 are shown.

```
With[{ε = 10^-12},
Show[GraphicsArray[Show[Table[(* use four sheets *)
  Plot3D[Evaluate[#[sheetg[j, k, x + I y]]], {x, -2, 2}, {y, ε, 2},
         PlotPoints -> {30, 15}, DisplayFunction -> Identity,
         ViewPoint -> {1, -3, 0.4}], {j, 0, 1}, {k, 0, 1}],
           DisplayFunction -> Identity]& /@ {Re, Im}]]]
```

Using $\frac{1}{z}$ instead of z makes the branch point from infinity visible at the origin.

```
With[{ε = 10^-12},
Show[GraphicsArray[Show[Table[(* use four sheets *)
  Plot3D[Evaluate[#[sheetg[j, k, 1/(x + I y)]]],
         {x, -2, 2}, {y, ε, 2},
         PlotPoints -> {31, 15}, DisplayFunction -> Identity,
         ViewPoint -> {1, -3, 0.4}], {j, 0, 1}, {k, 0, 1}],
           DisplayFunction -> Identity]& /@ {Re, Im}]]]
```

To get a unified view on the finite branch points as well as the one at infinity, we will construct a picture that does not show Im($g(z)$) or Re($g(z)$) over the complex z-plane, but rather over the Riemann sphere to cover all z-values more equally. Given the Riemann sphere of radius $R = 1/2$ around the point $\{0, 0, 1/2\}$, we visualize Im($g(x + i\,y)$) as a point in direction of the image of $x + i\,y$ on the Riemann sphere and distance radius $r = R + r\arctan(\text{Im}(g(x+i\,y)))$. We use the arctan in the last formula because it allows us to uniquely map the interval $(-\infty, \infty)$ to a finite interval. The function `sphereSheetg` calculates the projections of the sheets onto the Riemann sphere.

```
sphereSheetg[j_, k_, φ_, ϑ_] :=
Module[{x, y, dir},
       {x, y} = Cot[ϑ/2] {Cos[φ], Sin[φ]};
       dir = {Cos[φ] Sin[ϑ], Sin[φ] Sin[ϑ], Cos[ϑ]};
       {0, 0, 1/2} +
       (* in radial direction *) dir (1/2 + 1/(2 Pi) *
                         ArcTan[Im[sheetg[j, k, x + I y]]])]
```

Here is one half of the resulting Riemann sphere surface. The branch point at infinity is now clearly visible at the north pole. The two branch points ± 1 are now at the equator.

```
ε = 10^-4;
Show[Graphics3D[
Table[{(* color sheets differently *)
       SurfaceColor[Hue[j/3 + k/2]], EdgeForm[{Thickness[0.001]}],
Cases[ParametricPlot3D[sphereSheetg[j, k, φ, ϑ],
                {φ, ε, Pi - ε}, {ϑ, ε, Pi - ε},
                PlotPoints -> 30, Compiled -> False,
                DisplayFunction -> Infinity],
       _Polygon, Infinity]}, {j, 0, 1}, {k, 0, 1}]],
     ViewPoint -> {-0.8, -3, 0.3}]
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

g) Without branch cuts, the function $f(z)$ is just the identity function. (The function `PowerExpand` does just this, ignoring branch cuts—we will discuss it in Chapter 1 of the Symbolics volume [67✱].)

```
1/Log[Exp[1/z]] // PowerExpand
```

But a plot along a line just above the real axis shows a much more complicated behavior. The outermost constant behavior is the one to be expected from $f(z) = z$.

```
Plot[Im[1/Log[Exp[1/(x + I 0.0015)]]], {x, -0.05, 0.05},
     PlotRange -> All, Axes -> False, Frame -> True]
```

`Exp` is a meromorphic function. So all branch cuts of $f(z)$ are caused by the branch cut of the `Log` function. Thus, the branch cuts of $f(z)$ are located where $f(z) = \textit{negativeRealNumber}$. The function `1/Log[Exp[z]]` has a countable infinite number of branch cuts parallel to the real axis at values $\text{Im}(z) = (2\,k + 1)\,\pi,\ k \in \mathbb{Z}$. By the inversion principle, $z \to \frac{1}{z}$ maps the straight lines into circles with midpoints $1/(2\,(2\,k+1)\,\pi)$ and radius $|1/(2\,(2\,k+1)\,\pi)|$. The following function `graph` visualizes this.

```
graph[lx_, ly_] :=
Module[{pp = 100, cs = 80},
Show[GraphicsArray[{
(* 3D plot *)
Plot3D[Im[1/Log[Exp[1/(x + I y)]]], {x, -lx, lx}, {y, -ly, ly},
             ColorFunction -> Hue, BoxRatios -> {1, ly/lx, 0.6},
             PlotPoints -> pp, Mesh -> False, ViewPoint -> {0, -2, 1.6},
             Axes -> {True, True, False}, DisplayFunction -> Identity],
(* contourplot *)
ContourPlot[Im[1/Log[Exp[1/(x + I y)]]], {x, -lx, lx}, {y, -ly, ly},
            ColorFunction -> (Hue[2 #]&), PlotPoints -> pp,
            Contours -> cs, ContourStyle -> {Thickness[0.001]},
            AspectRatio -> ly/lx, DisplayFunction -> Identity],
(* pole location graphics *)
Graphics[{Thickness[0.001],
          Table[Circle[{0, 1/(2 k + 1)/Pi/2}, Abs[1/(2 k + 1)/Pi/2]],
                {k, -Floor[1/ly] - 10, Floor[1/ly] + 10}]},
          PlotRange -> {{-lx, lx}, {-ly, ly}},
          AspectRatio -> ly/lx, Frame -> True]}]]]
```

The left picture shows a 3D plot of the imaginary part of $f(z)$. The branch cuts appear as steep walls in this picture. The middle graphic shows a contour plot of the imaginary part of $f(z)$. This time the branch cuts are visible as clusters of contour lines. And the right picture shows circles with midpoints $1/(2(2k+1)\pi)$ and radius $|1/(2(2k+1)\pi)|$ for comparison.

```
graph[0.3, 0.5]
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

h) The branch points and the branch cuts of `ArcCoth` follow uniquely from the branch points and branch cuts of the `Log` function.

The function $z \to 1 - 1/z$ maps the branch points 0 and ∞ to 1 and 0. $1 - 1/z$ is negative for $z \in (0, 1)$. Similarly, the function $z \to 1 + 1/z$ maps the branch points 0 and ∞ to -1 and 0. $1 + 1/z$ is negative for $z \in (-1, 0)$. This means the points $z = \pm 1$ will surely be logarithmic branch points. The two logarithmic branch points that are mapped to 0 basically cancel each other. Because the two functions $z \to 1 \pm 1/z$ map the negative real line "from different directions", the only surviving feature of the canceling branch points at $z = 0$ is a discontinuity for arccoth(x) along the real axis at $x = 0$. As a result, we have near the origin $\text{arccosh}(x) \propto \pm i\pi/2 + x + O(x)^3$. Because the branch cut along $(-1, 1)$ is solely caused from the logarithm, the absolute value of the jump size will be $|\pi|$ along the whole branch cut.

Here are pictures of Im(arccosh(z)) along the real line and over the complex z plane. The rightmost picture shows a part of the Riemann surface of arccoth(z), by displaying Im(arccoth(z)).

```
pictures[function_, continuedFunctions_, vp1_, vp2_] :=
Show[GraphicsArray[
Module[{regions, ε = 10^-($MachinePrecision - 2)},
regions =  (* subdivide z-plane to avoid branch cuts *)
 {{{0, 1 - ε}, {ε, 3/2}}, {{0, -1 + ε}, {ε, 3/2}},
  {{0, 1 - ε}, {-ε, -3/2}}, {{0, -1 + ε}, {-ε, -3/2}},
  {{1 + ε, 2}, {ε, 3/2}}, {{-2, -1 - ε}, {ε, 3/2}},
  {{1 + ε, 2}, {-ε, -3/2}}, {{-2, -1 - ε}, {-ε, -3/2}}};
(* the three graphics *)
Block[{$DisplayFunction = Identity},
{(* imaginary part along the real axis *)
 Plot[Im[function[x]], {x, -2, 2}, PlotStyle -> {Thickness[0.01]}],
 (* imaginary part over the complex plane *)
 Show[Apply[Plot3D[Im[function[x + I y]],
     Evaluate[{x, Sequence @@ #1}, {y, Sequence @@ #2}],
       PlotPoints -> 20, Mesh -> False]&, regions, {1}],
      ViewPoint -> vp1],
(* some sheets of the Riemann surface *)
Show[Show[Function[f, Apply[Plot3D[f,
    Evaluate[{x, Sequence @@ #1}, {y, Sequence @@ #2}],
             PlotPoints -> 20, Mesh -> False]&,
    regions, {1}]] /@ continuedFunctions], Boxed -> True,
    Axes -> False, BoxRatios -> {1, 1, 1.4}, ViewPoint -> vp2]}]],
   GraphicsSpacing -> -0.02]]

(* the three graphics for ArcCoth *)
pictures[ArcCoth, Flatten[Table[
  {Im[(Log[1 + 1/(x + I y)] + k1 2 I Pi -
      Log[1 - 1/(x + I y)] + k2 2 I Pi)/2]},
   {k1, -1, 1}, {k2, -1, 1}]], {1, 3, 1.6}, {-1, 3, 1.2}]
```

The two sides of the two branch cuts form two (locally) disconnected pieces of the Riemann surface of arccoth(z) in the interval $(-1, 1)$. This means that encircling the origin with a radius < 1 yields after one round the same function value as before. Using a radius > 1 we enclose the two logarithmic branch points and after one round we come back to the starting point (such a contour can be viewed as encircling infinity and shows that infinity is not a branch point of arccoth—at $z = \infty$ we have the expansion $\operatorname{arccoth}(z) \propto z^{-1} + z^{-3}/3 + O(z^{-1})^4$). Moving around any of the two logarithmic branch points with a radius < 1 brings one to another sheet of the Riemann surface and the function value changes by $\pm i\pi$. Repeatedly encircling any of the two logarithmic branch points brings one to ever-new sheets of the Riemann surface of arccoth(z). Moving along the eight-shaped contour $\{2\cos(\varphi), \sin(\varphi)\}$ “skips” every second sheet and after one round the function value has changed by $\pm 2 i\pi$. The picture above of the Riemann surface of arccoth(z) lets us easily verify the above considerations.

The branch points and the branch cuts of `ArcCosh` follow from the branch points and branch cuts of the `Sqrt` and the `Log` function. The arguments of the two `Sqrt` functions taken separately generate the two branch points ± 1 and the branch cuts are $(-\infty, -1]$ and $(-\infty, 1]$. This means that in the interval $(-\infty, -1]$ two branch cuts coincide. In this interval, they actually cancel leaving the interval $[-1, 1]$ as the branch cut of $z + \sqrt{z-1}\,\sqrt{z+1}$. Nowhere in the complex plane does the argument of the logarithm $z + \sqrt{z-1}\,\sqrt{z+1}$ assume the value 0. This means that this branch point of the `Log` function is absent in the principal sheet of arccosh. For $z \to -\infty$, the argument of the logarithm approaches $-\infty$ and we have a logarithmic branch point there. Although the argument 0 branch point is not present, the branch cut of the logarithmic function is still there. For $z < -1$, the argument of the logarithm is negative real and as a result, in the interval $(-\infty, -1]$ we have a branch cut caused by the logarithm function. This means that for $z < -1$ the value of the jump height is $|2\pi|$. In the interval $(-1, 1)$ it is $|2\operatorname{arccos}(x)|$.

Similar to the arccoth function, encircling the origin with a radius < 1 yields after one round the same function value as before. Around the origin, we have the expansion $\operatorname{arccosh}(z) = \pm i\,\pi/2 \pm i\,z + O(z)^2$. Using a radius greater than 1, we enclose the two square root branch points. At the same time, such a contour encloses the logarithmic branch point at $-\infty$ and as a result, the value changes by $\pm 2\,\pi\,i$. ($\operatorname{arccosh}(z)$ can be approximated by $\log(-4\,z^2)/2{=}\pi\,\sqrt{-z^2}\,\Big/ z - z^{-2}/4 + O(z^{-1})^3$ at infinity.) Moving around any of the two square root branch points brings with a radius < 1 brings us to the other sheet of the Riemann surface and after two revolutions, we return. Moving along the eight-shaped contour $\{2\cos(\varphi), \sin(\varphi)\}$ also causes the function value to change by $\pm 2\,i\,\pi$.

The following pictures of the principal value of $\operatorname{arccosh}(z)$ and the Riemann surface of $\operatorname{arccosh}(z)$ lets us easily visualize the above considerations.

```
(* the three graphics for ArcCosh *)
pictures[ArcCosh, Flatten[Table[
  {Im[Log[(x + I y) + k1 Sqrt[-1 + (x + I y)]*
          Sqrt[1 + (x + I y) ]] + k2 2 I Pi]},
   {k1, -1, 1, 2}, {k2, -1, 1}]], {1, -3, 1.6}, {-1, 3, 0.9}]
```

The branch points and the branch cuts of `ArcSech` follow from the branch points and branch cuts of the `Sqrt` and the `Log` function. The arguments of the two `Sqrt` functions generate the two branch points ± 1 and 0. The function $z \to 1/z - 1$ is negative for $z \in (-\infty, 0) \bigvee (1, \infty)$ and the function $z \to 1/z + 1$ is negative for $z \in (-1, 0)$. This means in the intervals $(-\infty, -1)$, $(1, \infty)$ we have branch cuts due to the `Sqrt` function. In the interval $(-1, 0)$, the two square root branch cuts cancel. For small arguments, the argument of the logarithm $1/z + \sqrt{1/z - 1}\,\sqrt{1/z + 1}$ can be approximated as $2\,z^{-1} - z/2 + O(z)^3$. This means that at $z = 0$ we have the "infinity branch point" of the logarithm. Nowhere does the argument of the logarithm vanish on the principal sheet and so the "zero branch point" of the logarithmic function does not exist on the principal sheet. In the interval $[-1, 0]$, the argument of the logarithm is negative real and so we have a jump height of $|\,2\,\pi\,|$ in this interval. In the intervals $(-\infty, 1]$ and $[1, \infty)$ the branch cuts are the ones of the square root functions and the jump height is $|\,2\operatorname{arcsech}(x)\,|$.

Encircling the origin with a radius less than 1 yields after one round a function value change of $\pm 2\,\pi\,i$. Using a radius greater than 1 yields the same function value. Infinity is not a branch point for $\operatorname{arcsech}(z)$. (At infinity, we have $\operatorname{arcsech}(z) = i\left(\pi/2 - 1/z + O(z^{-1})^3\right)$.) Moving around any of the two square root branch points with a radius less than 1 brings us to another sheet of the Riemann surface and after two revolutions, the starting function value is obtained again. Moving along the eight-shaped contour $\{2\cos(\varphi), \sin(\varphi)\}$ is not possible here because the path would go through the logarithmic branch point at the origin.

The following picture of the principal value of $\operatorname{arcsech}(z)$ and the Riemann surface of $\operatorname{arccosh}(z)$ lets us again easily verify the above considerations.

```
(* the three graphics for ArcSech *)
pictures[ArcSech, Flatten[Table[
  {Im[Log[k1 Sqrt[1/(x + I y) + 1] Sqrt[1/(x + I y) - 1] +
                                    1/(x + I y)] + 2Pi I k2]},
             {k1, -1, 1, 2}, {k2, -1, 1}]],
        {1, 3, 1.6}, {-2, 3, 0.5}]
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

7. “Strange” Analytic Functions

a) To construct a discontinuous function that is 1 on a 1D sub-manifold of the complex numbers and 0 everywhere from analytic functions, we obviously need a function that has a branch cut. Using the continuity of such a function from one side, we have to arrange that two branch cuts overlap and have continuity from different sides. Here is one possible choice for such a function.

```
f1[z_] = (Log[z] + Log[1/z])/(2 I Pi)
```

The function `f1` is zero almost everywhere in the complex z-plane. It is 1 along the negative real axis. Plots confirm this fact. (We multiply the function values by $10^{machinePrecision}$.)

```
Plot3D[Evaluate[10^$MachinePrecision Abs[f1[x + I y]]],
       {x, -3, 3}, {y, -3, 3}, PlotPoints -> 20]

Plot[Abs[f1[x]], {x, -3, 3},
     PlotRange -> All, Frame -> True, Axes -> False]
```

To get a function that is 1 on the unit circle, we map the negative real line onto the unit circle using $z \to \log(z)/i - 2\pi$.

```
f2[z_] = f1[Log[z]/I - 2Pi]
```

Here is a graphic of `f2` over the complex z-plane. (We multiply the function values by $10^{machinePrecision}$.)

```
Plot3D[Evaluate[10^$MachinePrecision Abs[f2[x + I y]]],
       {x, -3, 3}, {y, -3, 3}, PlotPoints -> 30]
```

On the unit circle, the function is 1. We use the function `Simplify` to show this property symbolically.

```
Table[f2[Exp[I φ]], {φ, 0, 2Pi, 2Pi/12}] // Simplify
```

Outside the unit circle, the function is 0. We use the function `FullSimplify` to show this property symbolically.

```
Table[f2[999/1000 Exp[I φ]], {φ, 0, 2Pi, 2Pi/12}] // FullSimplify

Table[f2[1001/1000 Exp[I φ]], {φ, 0, 2Pi, 2Pi/12}] // FullSimplify
```

At $z = 0$, the function `f2` is not defined.

```
f2[0]
```

No other finite value z exists where `f2[z]` is undefined. For this to happen, `-2 Pi - I Log[z]` must vanish, which is not possible.

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

b) As an initial step, it is straightforward to construct a function that vanishes in the whole left-side plane and is 1 in the right-hand plane. Here is such a function.

```
f1[z_] = 1 + (Sqrt[z^2] - z)/(2z)
```

At $z = 0$, the function `f1` is undefined.

```
f1[0]

Plot3D[Abs[f1[x + I y]], {x, -12, 12}, {y, -12, 12},
       PlotPoints -> 50, PlotRange -> All]
```

Using the conformal map $z \to 1/(z + 1) - 1/2$, we can map the right-hand plane onto the unit disk.

```
f2[z_] = f1[1/(z + 1) - 1/2]
```

The resulting function `f2` has the desired property to vanish outside the unit circle and be 1 inside the unit circle. The next graphic shows the real and the imaginary part of `f2` over the complex z-plane. The imaginary part shows fluctuations caused by differences in the last digit of machine numbers of size 10^0.

```
Show[GraphicsArray[
Function[{reIm},
Plot3D[reIm[f2[x + I y]], {x, -3, 3}, {y, -3, 3},
       PlotPoints -> 120, PlotRange -> All, Mesh -> False,
       DisplayFunction -> Identity]] /@
       (* real and imaginary part *) {Re, Im}]]
```

On the unit circle, the function `f2` has two undefined points, $z = \pm 1$; it is 0 on the upper half of the unit circle and 1 on the lower half.

```
(* avoid messages *)
Off[Power::infy]; Off[Infinity::indet]; Off[N::meprec]
Table[f2[Exp[I φ]], {φ, 0, 2Pi, 2Pi/12}] // N[#, 22]&
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

c) Similarly to the last two problems, we will make use of the branch cuts of analytic functions. Let us start to build a function that is 1 at $z = 0$ and 0 almost everywhere else. The following function is 1 on the negative real axis.

```
f1[x_] = (Sqrt[x] - 1/Sqrt[1/x])/(2 I Sqrt[-x]);
```

```
Plot[Abs[f1[x]], {x, -5, 5},
     PlotRange -> All, Frame -> True, Axes -> False,
     PlotStyle -> {Thickness[0.01]}]
```

It is zero everywhere else.

```
{f1[4], f1[0.1], f1[-N[3, 22] - I]}
```

The next function `f2` does not vanish anywhere and is negative along the negative real axis.

```
f2[x_] = x + Sqrt[x - 2] Sqrt[x] - 1;
```

```
Plot[Re[f2[x]], {x, -5, 5},
     PlotRange -> All, Frame -> True, Axes -> False,
     PlotStyle -> {Thickness[0.01]}]
```

At the point where the real part of `f2` vanishes, its imaginary part does not.

```
f2[1]
```

Using `f1` and `f2`, we can build a function `f3` that is 1 at $z = 0$, and 0 everywhere else on the real axis.

```
f3[z_] = f1[f2[z]] + f1[f2[-z]] - 1;
```

```
f3[0]
```

```
Plot[Abs[f3[x]], {x, -5, 5},
     PlotRange -> {-1, 1}, Frame -> True, Axes -> False,
     PlotStyle -> {Thickness[0.01]}]
```

Here is another possibility for a function with the required properties.

```
f4[x_] = f1[I x - 3]
```

```
f4[0]
```

```
Plot[Abs[f4[x]], {x, -5, 5},
     PlotRange -> {-1, 1}, Frame -> True, Axes -> False,
     PlotStyle -> {Thickness[0.01]}]
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

d) The function `f1` is 1 along negative real axis.

```
f1[x_] = (Sqrt[x] - 1/Sqrt[1/x])/(2 I Sqrt[-x]);

Plot[Abs[f1[x]], {x, -3, 3},
     PlotRange -> All, Frame -> True, Axes -> False,
     PlotStyle -> {Thickness[0.01]}]
```

The function `f2` is 1 everywhere on the real axis (with the exception of 0, a point we will exclude later).

```
f2[x_] = f1[-x^2];

Plot[Abs[f2[x]], {x, -3, 3},
     PlotRange -> {0, 2}, Frame -> True, Axes -> False,
     PlotStyle -> {Thickness[0.01]}]
```

Using now the function `f3` which does not vanish anywhere, we can construct the function `f4` with the required property.

```
f3[x_] = x + Sqrt[x - 2] Sqrt[x] - 1;

f4[x_] = 1 - f2[f3[2x]];

{f4[0], f4[1]}

Plot[Abs[f4[x]], {x, -3, 3},
     PlotRange -> All, Frame -> True, Axes -> False]
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

e) This is a graph of the function to be modeled.

```
Plot[Floor[x], {x, -4, 4}, PlotStyle -> {Thickness[0.01]}]
```

A function that is stepwise constant is, for instance, $x - \tan^{(-1)}(\tan(x))$.

```
Plot[x - ArcTan[Tan[x]], {x, -8, 8},
     PlotStyle -> {Thickness[0.01]}]
```

Adjusting the step size and the step height of the last function leads to the function $x + \tan^{-1}(\cot(\pi\, x)) \big/ \pi - 1/2$. Its graph coincides with the graph of $\lfloor x \rfloor$.

```
f[x_] := x + ArcTan[Cot[Pi x]]/Pi - 1/2

Plot[f[x], {x, -4, 4}, PlotStyle -> {Thickness[0.01]}];
```

At integer values, the function `f` is ill defined.

```
{f[-2], f[-1], f[0], f[1], f[2]}
```

For similar expressions for $\lfloor x \rfloor$, see [62✱].

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

f) This is a graph of the function to be modeled.

```
Plot[IntegerPart[Mod[x, 2]], {x, -4, 4},
     PlotRange -> All, PlotStyle -> {Thickness[0.01]}]
```

A function that is stepwise constant is, for instance, $\sqrt{\sin^2(x)} \Big/ \sin(x)$.

```
Plot[Sqrt[Sin[x]^2]/Sin[x], {x, -8, 8},
     PlotStyle -> {Thickness[0.01]}]
```

Adjusting the step size and the step height of the last function leads to the function $\left(1 - \left(\sin^2(\pi\, x)\right)^{1/2} \Big/ \sin(\pi\, x)\right) \Big/ 2$. Its graph coincides with the graph of $x \bmod 2$.

```
f[x_] := (1 - Sqrt[Sin[Pi x]^2]/Sin[Pi x])/2
```

```
Plot[(1 - Sqrt[Sin[Pi x]^2]/Sin[Pi x])/2, {x, -4, 4},
     PlotRange -> All, PlotStyle -> {Thickness[0.01]}]
```

At integer values, the function `f` is ill-defined.

```
{f[-2], f[-1], f[0], f[1], f[2]}
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

g) This is a graph of the function to be modeled.

```
Plot[(1 - 2 Abs[Round[x/2] - x/2]), {x, -4, 4},
     PlotRange -> All, PlotStyle -> {Thickness[0.01]}]
```

A sawtooth function, that is, for instance, $\sin^{(-1)}(\sin(x))$.

```
Plot[ArcSin[Sin[x]], {x, -8, 8},
     PlotStyle -> {Thickness[0.01]}]
```

Adjusting the step size and the step height of the last function leads to the function $\left(\sin^{-1}(\sin(\pi x + \pi/2)) + \pi/2\right)/\pi$. Its graph coincides with the graph of $x \bmod 2$.

```
f[x_] := (ArcSin[Sin[Pi x + Pi/2]] + Pi/2)/Pi
```

```
Plot[f[x], {x, -4, 4},
     PlotRange -> All, PlotStyle -> {Thickness[0.01]}]
```

The two functions also agree at the nondifferentiable points.

```
{f[-2], f[-1], f[0], f[1], f[2]}
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

8. `ArcTan[(x + 1)/y]- ArcTan[(x - 1)/y]` Picture

Here is the function to be displayed.

```
f[x_, y_] := ArcTan[(x + 1)/y] - ArcTan[(x - 1)/y]
```

```
ε = 10^-14;
```

```
Plot3D[Evaluate[f[x, y]], {x, -Pi, Pi}, {y, ε, Pi},
      PlotPoints -> 50, PlotRange -> All]
```

`ArcTan` is a smooth function for real-valued `xy`, but as `xy` ⟶ `±Infinity`, it approaches the different limiting values `±Pi`.

```
Plot[ArcTan[xy], {xy, -5, 5}, PlotStyle -> {Thickness[0.01]}]
```

```
Plot[ArcTan[1/xy], {xy, -5, 5}, PlotStyle -> {Thickness[0.01]}]
```

Now, let us look at `x/y`. For small y in $(-\varepsilon,\ \varepsilon)$, the argument becomes large and changes sign, so we have a jump at $y = 0$.

```
Plot3D[ArcTan[x/y], {x, -Pi, Pi}, {y, -Pi, Pi},
      PlotPoints -> 30]
```

```
Show[Plot3D[ArcTan[x/y], {x, -Pi, Pi}, #,
           PlotPoints -> {60, 30}, DisplayFunction -> Identity]& /@
           {{y, ε, Pi}, {y, -ε, -Pi}},
     PlotRange -> All, DisplayFunction -> $DisplayFunction]
```

Now let us look at the difference `ArcTan[(x + 1)/y] - ArcTan[(x - 1)/y]`. Taking the above into account means that for $x > 1$ and $x < -1$, the two jumps cancel, and for $-1 < x < 1$, they add. For $y > 0$, $y \to 0$, they add to π, and for $y < 0$, $y \to 0$, they add to $-\pi$.

```
Show[Plot3D[Evaluate[f[x, y]], {x, -Pi, Pi}, #,
            PlotPoints -> {60, 30},
            DisplayFunction -> Identity]& /@
             {{y, 10^-14, Pi}, {y, -10^-14, -Pi}},
     PlotRange -> All, DisplayFunction -> $DisplayFunction]
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

9. `ArcSin[ArcSin[z]]` Picture

The function `ArcSin` has two branch points at +1 and -1.

```
Plot3D[Im[ArcSin[x + I y]],
       {x, -3, 3}, {y, -3, 3}, PlotPoints -> 40]

Plot3D[Re[ArcSin[x + I y]],
       {x, -3, 3}, {y, -3, 3}, PlotPoints -> 40]
```

This means the function `ArcSin[ArcSin[z]]` will have branch points at $+1$ and -1 as well, and in addition at the points where `ArcSin[z]=±1`. This is the case at `z=±ArcSin[1]`.

```
{Sin[1], Sin[-1]} // N
```

Here, the resulting picture is shown.

```
Plot3D[Im[ArcSin[ArcSin[x + I y]]], {x, -3, 3}, {y, -3, 3},
       PlotPoints -> 40]
```

In this picture, the original branch points at ±1 are not easy to recognize. Zooming in a bit, we see them more clearly.

```
Plot3D[Im[ArcSin[ArcSin[x + I y]]], {x, 0.7, 1.2}, {y, -0.1, 0.1},
       PlotPoints -> 40]
```

Following the imaginary part just above the real axis, we see the original branch points quite pronounced.

```
Plot[Im[ArcSin[ArcSin[x + I 10^-12]]], {x, 0.7, 1.2},
     AxesOrigin -> {0.7, 0}, PlotStyle -> {Thickness[0.006]}]
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

10. Singularities of tanh(sinh(cot(z))), exp$\left(\ln^{i\pi}(z)\right)$ Properties

a) The following picture shows a coarse contour plot of the real part of the function $w(z)$. The periodicity $w(z) = w(z + \pi)$ caused by the innermost cot function is clearly visible.

```
w[z_] = Tanh[Sinh[Cot[z]]];

(* suppress messages arising from trying to calculate very small
   and very large numbers *)
Off[General::ovfl]; Off[General::unfl];
ContourPlot[Re[w[x + I y]], {x, -Pi, Pi}, {y, -1, 1},
            Contours -> 20, PlotPoints -> 120, ContourLines -> False,
            ColorFunction -> (Hue[0.8 #]&)]
```

The singularities of the innermost cot function are poles of order 1 at $z = k\,\pi$. (We will discuss the function `Series` in Chapter 1 of the Symbolics volume [67✱].)

```
Series[Cot[z], {z, 0, 2}]
```

Sinh does not have singularities by itself. The cot poles become essential singularities. (This becomes obvious when recalling the identity $\sinh(z) = (e^z - e^{-z})/2$.) In a contour graphic, essential singularities of the type $\exp(1/z)$ show a typical flower-like form.

```
ContourPlot[Re[Sinh[1/(x + I y)]], {x, -0.5, 0.5}, {y, -0.3, 0.3},
            Contours -> 50, PlotPoints -> 250, ContourLines -> False,
            ColorFunction -> (Hue[0.8 #]&), AspectRatio -> Automatic]
```

The outer function tanh has singularities at $z = i(\pi/2 + k\,\pi),\;\; k \in \mathbb{Z}$. The singularities are again poles of order 1.

```
Series[Tanh[z], {z, I Pi/2, 2}]
```

In a contour plot, poles of order 1 are visible as nested eight-shaped regions.

```
ContourPlot[Re[Tanh[x + I y]], {x, -5, 5}, {y, -8, 8},
            PlotPoints -> 120, ContourLines -> False,
            ColorFunction -> (Hue[0.8 #]&), Contours -> 20]
```

The essential singularities at $z = k\,\pi$ stay essential singularities under the mapping $z \to \tanh(z)$. In addition, the mentioned first-order poles from tanh appear as singularities. They are located at $\sinh(\cot(z)) = i(\pi/2 + k\,\pi),\;\; k \in \mathbb{Z}$. Solving the last equation for the position of these poles by inversion, and taking into account the symmetry and periodicity of sinh and cot, gives $z = \operatorname{arccot}(\operatorname{arcsinh}(i(\pi/2 + k\,\pi) +\; i\,l\,\pi)) + m\,\pi,\; k,\;\; l,\;\; m\;\in \mathbb{Z}$. The terms $m\,\pi$ represent the periodicity along the real axis. The double infinite set of poles indexed by k and l lie along contours that form the flower-shaped essential singularities and cluster at the essential singularities. The following graphic shows again a contour plot of $w(z)$ (this time we use the imaginary part) together with the location of some of the poles (indicated as crosses).

```
somePoles = Flatten[Table[ArcCot[ArcSinh[I(Pi/2 + k Pi)] + I l Pi],
                             {k, -10, 10}, {l, -10, 10}] // N, 1];

makeCross[z_] := (* a small cross *)
Module[{x = Re[z], y = Im[z], l = 0.01},
       {Line[{{x, y - l}, {x, y + l}}],
        Line[{{x - l, y}, {x + l, y}}]}]

ContourPlot[Im[w[x + I y]], {x, 0, 0.5}, {y, 0.02, 0.5},
            Contours -> 50, ContourLines -> False,
            PlotPoints -> 120, ColorFunction -> (Hue[3 #]&),
            (* plot crosses on top *)
            Epilog -> {Thickness[0.001], GrayLevel[0],
                       makeCross /@ somePoles}]
```

Along the real axis, the function $w(x)$ has the interesting property that the derivatives of all orders vanish when approaching the singularities. As a result, the graph of the function is nearly parallel to the x-axis.

```
Plot[w[x], {x, 0, Pi}, Frame -> True, Axes -> False]
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

b) The branch cuts of the function $f(z)$ arise from the branch cuts of the functions ln and power. The inner logarithm gives rise to a branch cut along the interval $(-\infty, 0]$. The function $g(z) = z^{i\pi} = \exp(i\,\pi\ln(z))$ again has a branch cut along the interval $(-\infty, 0]$. This means that $\ln^{i\pi}(z)$ has an additional branch cut along the interval $[0, 1]$.

Let $z = |z|\,e^{i\varphi}$. Then we have the following identities for the absolute value and the argument of the functions ln, power, and exp.

$$\ln(z) = \sqrt{\varphi^2 + \log^2(|z|)}\;\exp(i\arctan(\log(|z|),\,\varphi))$$

$$z^{i\pi} = e^{-\pi\varphi} \exp(i\,(\cos(\pi \log(|z|)), \sin(\pi \log(|z|))))$$

$$\exp(z) = e^{|z|\cos(\varphi)} \exp(i \arctan(\cos(|z| \sin(\varphi)), \sin(|z| \sin(\varphi))))$$

Putting the last formulas together leads to the following expression for $\arg\left(\exp\left(\ln^{i\pi}(z)\right)\right)$.

$$\arg\left(\exp\left(\ln^{i\pi}(z)\right)\right) = \\ \arctan\left(\cos\left(e^{-\pi\arctan(\log(|z|),\varphi)} \sin\left(\arctan\left(\cos\left(\pi \log\left(\sqrt{\varphi^2 + \log^2(|z|)}\right)\right), \sin\left(\pi \log\left(\sqrt{\varphi^2+\log^2(|z|)}\right)\right)\right)\right)\right), \\ \sin\left(e^{-\pi\arctan(\log(|z|),\varphi)} \sin\left(\arctan\left(\cos\left(\pi \log\left(\sqrt{\varphi^2 + \log^2(|z|)}\right)\right), \sin\left(\pi \log\left(\sqrt{\varphi^2+\log^2(|z|)}\right)\right)\right)\right)\right)\right)$$

The dominating term of the last expression is $\exp(-\pi \arctan(\log(|z|), \varphi))$. For $0 \lesssim x \lesssim 1$ and $0 \lesssim y \lesssim -\pi$, this expression takes on large values compared to the other expressions that are bounded by ± 1. Here this is visualized.

```
g[r_, φ_] = Exp[-Pi ArcTan[Log[r], φ]];

Show[GraphicsArray[{Show[#], Show[#, ViewPoint -> {4, 0.4, 1}]}&[
Plot3D[g[Sqrt[x^2 + y^2], ArcTan[x, y]], {x, -1/2, 3/2}, {y, 1, -1},
       PlotPoints -> 200, Mesh -> False, PlotRange -> All,
       AspectRatio -> Automatic, DisplayFunction -> Identity]]]]
```

The large values of $g(r, \varphi)$ in $\arctan(\cos(g(r, \varphi)\, h(r, \varphi)), \cos(g(r, \varphi)\, h(r, \varphi)))$ where $h(r, \varphi)$ is the following bounded function with an oscillating behavior near $z = 1$ causes most of the structure in $\arg(f(z))$.

```
h[r_, φ_] := Sin[ArcTan[Cos[Pi Log[Sqrt[Log[r]^2 + φ^2]]],
                        Sin[Pi Log[Sqrt[Log[r]^2 + φ^2]]]]]

Plot3D[h[Sqrt[x^2 + y^2], ArcTan[x, y]], {x, -2, 2}, {y, 1, 0},
       PlotPoints -> 200, Mesh -> False, PlotRange -> All,
       AspectRatio -> Automatic]

(* definition for f(z) *)
f[z_] := Exp[Log[z]^(Pi I)]

(* for z == 1/2 I f[z] agrees with above definition *)
{Arg[f[-1/2I]], ArcTan[Cos[g[1/2, -Pi/2] h[1/2, -Pi/2]],
                       Sin[g[1/2, -Pi/2] h[1/2, -Pi/2]]]} // N
```

The large values of $g(r, \varphi)$ result in $\arg\left(\exp\left(\ln^{i\pi}(z)\right)\right)$ being a highly oscillating function inside the rectangle $0 \lesssim x \lesssim 1$ and $0 \lesssim y \lesssim -1/2$. The following function `argPlot` shows a density plot of $\arg\left(\exp\left(\ln^{i\pi}(z)\right)\right)$ in the square $\{-L, L\} \times \{-L, L\}$.

```
argPlot[L_, opts___] :=
DensityPlot[Arg[Exp[Log[x + I y]^(Pi I)]], {x, -L, L}, {y, -L, L},
            opts, PlotPoints -> 500, ColorFunction -> Hue,
            PlotRange -> {-Pi, Pi}, Mesh -> False,
            FrameTicks -> None]
```

Because of the logarithmic singularity along the real axis, for smaller and smaller L we obtain qualitatively similar pictures despite L varying over many orders of magnitude.

```
Show[GraphicsArray[argPlot[#, DisplayFunction -> Identity]& /@
     {1, 10^-3, 10^-6}]]
```

For an analysis of the near $z \approx 0$ especially smooth function $w(z) = \exp\left(\ln^2(z)\right)$, see [55✱].

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

11. `Exp[-1/Im[1/(-Log[Infinity] + 2)^2]]`

The evaluation starts with log(∞); the result is `Indeterminate`.

```
Log[Infinity]
```

The next operation is $x \to 1 \big/ (x+2)^2$, which results in 0.

```
1/(-Infinity + 2)^2
```

The imaginary part of 0 is again 0.

```
Im[0]
```

$-1/0$ gives `ComplexInfinity` (the direction in the complex plane is not known).

```
-1/0
```

`Exp[ComplexInfinity]` finally gives `Indeterminate`.

```
Exp[ComplexInfinity]
```

Here, all calculations are carried out at once.

```
Exp[-1/Im[1/(-Log[Infinity] + 2)^2]]
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

12. Predict the Result

`Exp[I 2]` is a number of absolute value 1. This number gets multiplied by the rational number `(1 - 10^-21)`. The result is then calculated with 22 digits of precision.

```
N[(1 - 10^-21) Exp[I 2], 22]
```

The result is a number that has an absolute value less than 1.

```
Abs[%]
```

Raising this number to the power ∞ gives 0.

```
N[(1 - 10^-21) Exp[I 2], 22]^Infinity
```

In the second example, we multiply `Exp[I 2]` by `1 - 10^-23` and calculate again a 22-digit approximation of this number. (But this time, we would need at least 23 digits to recognize that the number has an absolute value less than 1.)

```
Abs[N[(1 - 10^-23) Exp[I 2], 22]]
```

Because, given the last number, it is not known if the number is slightly less, exactly equal to, or slightly larger than 1 in absolute value, the result of raising the number to power ∞ results in `Indeterminate`.

```
N[(1 - 10^-23) Exp[I 2], 22]^Infinity
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

13. $\tan(k/\alpha) + \tan(\alpha\, k)$ Picture

This is the definition of `tanPicture`.

```
tanPicture[α_] :=
ListPlot[Table[Tan[α k] + Tan[1/α k], {k, 20000}],
         PlotStyle -> {PointSize[0.001]}, Frame -> True,
         Axes -> False, FrameTicks -> None,
         PlotRange -> {-2, 2}, (* do not display single graphics *)
         DisplayFunction -> Identity]
```

Here is a list of different values of values for α that result in "qualitatively different" looking pictures. (The list is not complete; it just represents a semi-random selection.)

```
αList = {(* first row *)
         2.1450489763149355 10^-15, 7.00444977688695 10^-13,
         2.868659950132477 10^-10, -2.9894725892552067 10^-6,
         (* second row *)
         -0.0000352291969801675, 0.00008138, 0.0001697076, 0.000468,
         (* third row *)
         0.0011016, -0.00683519, -0.026557, 0.0296867,
         (* fourth row *)
         0.03296462, -0.05, -0.09825684, 0.245933,
         (* fifth row *)
         -0.8872561, 2.8614911, -3.28630, -35.7184,
         (* sixth row *)
         -4375.253, -221.335, -794.232, 1707.33,
         (* last row *)
         2405.25, 2701.76, -174277.21, -5.13315 10^8};
```

Here are the corresponding graphics.

```
Show[GraphicsArray[tanPicture /@ #]]& /@ Partition[αList, 4]
```

Be aware that these graphics are not always (mathematically) correct. Due to the use of machine arithmetic, some of the values of $\tan(k\,\alpha) + \tan(k/\alpha)$ are wrong. The next graphic shows the second of the above graphics calculated using high-precision arithmetic.

```
Show[tanPicture[N[700444977688695 10^-27, 30]],
     DisplayFunction -> $DisplayFunction]
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

References

✱1 L. V. Ahlfors. *Complex Analysis*, McGraw-Hill, New York, 1953. *BookLink*

✱2 J.-P. Allouche, J. Shallit. *Automatic Sequences*, Cambridge University Press, Cambridge, 2003. *BookLink*

✱3 J. Arndt, C. Haenel. *π Unleashed*, Springer-Verlag, Berlin, 2001. *BookLink*

✱4 H. Aslaksen. *SIGSAM Bull.* 30, n2, 12 (1996). *DOI-Link*

✱5 J. Beaumont, R. Bradford, J. H. Davenport in J. R. Sendra (ed.). *ISSAC 2003*, ACM Press, New York, 2003. *DOI-Link*

✱6 J. C. Beaumont, R. J. Bradford, J. H. Davenport, N. Phisanbut in J. Gutierrez (ed.). *ISSAC 04*, ACM Press, New York, 2004. *DOI-Link*

✱7 M. Beeson, F. Wiedijk in J. Calmet, B. Benhamou, O. Caprotti, L. Henocque, V. Sorge (eds.). *Artificial Intelli‑ gence, Automated Reasoning, and Symbolic Computation*, Springer-Verlag, Berlin, 2002. *BookLink*

✱8 H. Behnke, F. Sommer. *Theorie der analytischen Funktionen einer komplexen Veränderlichen*, Springer-Verlag, Berlin, 1962. *BookLink (2)*

✱9 L. Berggren, J. Borwein, P. Borwein. *Pi: A Source Book*, Springer-Verlag, New York, 1997. *BookLink (3)*

✱10 R. Bradford, R. M. Corless, J. H. Davenport, D. J. Jeffrey, S. M. Watt. *Ann. Math. Artif. Intell.* 36, 303 (2002). *DOI-Link*

✱11 R. J. Bradford, J. H. Davenport in T. Mora (ed.). *ISSAC 2002*, ACM, New York, 2002. *DOI-Link*

✱12 J. W. Bradshaw. *Ann. Math.* 4, 51 (1903).

✱13 I. N. Bronstein, K. A. Semandjajew. *Handbook of Mathematics*, Van Nostrand, New York, 1991. *BookLink*

✱14 J. H. E. Cartwright, D. L. González, O. Piro, D. Stanzial. *J. New Music Res.* 31, 51 (2002). *DOI-Link*

✱15 M. E. Catalan. *Nov. Ann. Math.* 8, 456 (1869).

✱16 P. S. Chee, S. T. Chin. *Coll. Math. J.* 11, 51 (1980).

✱17 H. Cohen, F. R. Villegas, D. Zagier. *J. Exp. Math.* 9, 3 (2000).

✱18 R. M. Corless, D. J. Jeffrey, S. M. Watt, J. H. Davenport. *SIGSAM Bull.* 34, n2, 58 (2000). *DOI-Link*

✱19 R. M. Corless, J. H. Davenport, D. J. Jeffrey, G. Litt, S. M. Watt in J. A. Campbell and E. Roanes–Lozano (eds.). *Artificial Intelligence and Symbolic Computation*, Springer-Verlag, Berlin, 2001.

✱20 J. H. Davenport in A. Asperti, B. Buchberger, J. H. Davenport (eds.). *Mathematical Knowledge Management 2003*, Springer-Verlag, Berlin, 2003. *BookLink*

✱21 J. H. Davenport in M. Joswig, N. Takayama (eds.). *Algebra, Geometry, and Software Systems*, Springer-Verlag, Berlin, 2003. *BookLink*

✱22 A. Dingle, R. J. Fateman in J. von zur Gathen, M. Giesbrecht (eds.). *Symbolic and Algebraic Computation*, ACM Press, New York, 1994. *DOI-Link*

✱23 A. J. Di Scala, M. Sombra. *arXiv:math.GM*/0105022 (2001). *Get Preprint*

✱24 D. G. Duffy. *Solving Partial Differential Equations*, CRC Press, Boca Raton, 1994. *BookLink (2)*

✱25 L. B. Felsen, I. N. Marcuvitz. *Radiation and Scattering of Waves*, IEEE Press, New York, 1994. *BookLink*

✱26 S. Finch. *Mathematical Constants*, Cambridge University Press, Cambridge, 2003. *BookLink*

✱27 J. Frauendiener, C. Klein. *J. Comput. Appl. Math.* 167, 193 (2004). *DOI-Link*

✱28 P. J. Grabner, T. Herendi, R. F. Tichy. *AAECC* 8, 33 (1997).

✱29 P. J. Grabner, H.-K. Hwang. *Constr. Approx.* 21, 149 (2005). *DOI-Link*

✱30 J. Havil. *Gamma*, Princeton University Press, Princeton, 2003. *BookLink*

✱31 P. Hertling. *Chaos, Solitons, Fractals* 10, 1087 (1999). *DOI-Link*

✱32 W. Kahan in A. Iserles, M. J. D. Powell (eds.). *The State of the Art in Numerical Analysis*, Clarendon Press, Oxford, 1987. *BookLink*

✱33 E. Kaplan. *The Nothing That Is*, Oxford University Press, Oxford, 1999. *BookLink (3)*

✱34 E. Karatsuba. *Num. Alg.* 24, 83 (2000). *DOI-Link*

✱35 D. E. Knuth. *Am. Math. Monthly* 99, 403 (1992).

✱36 G. A. Korn, T. M. Korn. *Mathematical Handbook for Scientists and Engineers*, McGraw-Hill, New York, 1968. *BookLink (2)*

✱37 L. Kuipers, H. Niederreiter. *Uniform Distribution of Sequences* Wiley, New York, 1974. *BookLink*

✱38 B. Lindström. *J. Number Th.* 65, 321 (1997). *DOI-Link*

✱39 G. D. Mahan. *Applied Mathematics*, Kluwer, New York, 2002. *BookLink*

✱40 E. Maor. *e: The Story of a Number*, Princeton University Press, Princeton, 1994. *BookLink (2)*

✱41 G. Markowsky. *Coll. Math. J.* 23, 3 (1992).

✱42 G. Markowsky. *Notices Am. Math. Soc.* 52, 344 (2005).

✱43 J. H. Mathews, R. W. Howell. *Complex Analysis for Mathematics and Engineering*, Jones and Bartlett, Boston, 1997. *BookLink (2)*

✱44 E. McClintock. *Am. J. Math.* 14, 72 (1891).

✱45 G. Melfi. *arXiv:math.NT*/0402458 (2004). *Get Preprint*

✱46 P. J. Nahin. *An Imaginary Tale*, Princeton University Press, Princeton, 1998. *BookLink*

✱47 R. Narasimhan, Y. Nievergelt. *Complex Analysis in One Variable*, Birkhäuser, Boston, 2001. *BookLink (2)*

✱48 S. S. Negi, R. Ramaswamy. *arXiv:nlin.CD*/0105011 (2001). *Get Preprint*

✱49 C. P. Niculescu, A. Vernescu. *J. Ineq. Pure Appl. Math.* 5, A55 (2004).

✱50 A. Olariu. *arXiv:physics*/9908036 (1999). *Get Preprint*

✱51 W. F. Osgood. *Lehrbuch der Funktionentheorie*, Teubner, Leipzig, 1923. *BookLink*

✱52 A. Ostrowski. *Abh. Hamburger Univ. Math. Sem.* 1, 77 (1929).

✱53 C. M. Patton. *SIGSAM Bull.* 30, n2, 21 (1996). *DOI-Link*

✱54 H. Pieper. *Die komplexen Zahlen*, Harry Deutsch, Frankfurt, 1988. *BookLink*

✱55 A. Pringsheim. *Math. Ann.* 50, 442 (1898).

✱56 I. K. Rana. *From Numbers to Analysis*, World Scientific, Singapore, 1998. *BookLink*

✱57 R. Remmert. *Funktionentheorie* v.1, v.2, Springer-Verlag, Berlin, 1992. *BookLink (2)*

✱58 B. Reznick. *J. Number Th.* 78, 144 (1999). *DOI-Link*

✱59 A. D. Rich, D. J. Jeffrey. *SIGSAM Bull.* 30, n2, 25 (1996). *DOI-Link*

✱60 J. F. Ritt. *Trans. Am. Math. Soc.* 27, 68 (1925).

✱61 C. Seife. *Zero*, Viking, New York, 2000. *BookLink (4)*

✱62 M. Sholander. *Am. Math. Monthly* 67, 213 (1960).

✱63 P. M. E. Shutler. *Int. J. Math. Edu. Sci. Technol.* 28, 677 (1997).

✱64 D. R. Stoutemyer. *Notices Am. Math. Soc.* 38, 778 (1991).

✱65 M. Trott. *The Mathematica GuideBook for Graphics*, Springer-Verlag, New York, 2004. *BookLink*

✱66 M. Trott. *The Mathematica GuideBook for Numerics*, Springer-Verlag, New York, 2005. *BookLink*

✱67 M. Trott. *The Mathematica GuideBook for Symbolics*, Springer-Verlag, New York, 2005. *BookLink*

✱68 A. J. van Zanten. *Nieuw Archief Wiskunde* 17, 229 (1999).

✱69 J. Vinson. *Exper. Math.* 10, 337 (2001).

✱70 R. Walser. *Der Goldene Schnitt*, Teubner, Stuttgart, 1993. *BookLink*

✱71 H. Zoladek. *Colloq. Math.* 84/85, 173 (2000).

CHAPTER 3

Definitions and Properties of Functions

3.0 Remarks

In this chapter, we discuss how to define simple functions in *Mathematica*. By simple, we mean simple in the form of their arguments. (Much more wide-ranging possibilities for defining functions will be presented in Chapter 5.) Definitions of recursive functions [46✱] and pure functions, along with attributes of functions, are important building blocks for the use of *Mathematica* to model arbitrary mathematical structures. Their applications range from simple to extremely complex.

```
(* no spelling warnings, set fonts for tick labels, ... *)
Get[ToFileName[ReplacePart["FileName" /.
 NotebookInformation[EvaluationNotebook[]], "Initialization.m", 2]]];
```

3.1 Defining and Clearing Simple Functions

■ 3.1.1 Defining Functions

It is essential to know when *Mathematica* is to carry out a symbolic operation, that is, whether a function is evaluated immediately when it is defined or only later when it is called. Indeed, if it is evaluated later, some of the values of the variables and functions involved in the right-hand side of the definition may have changed. Moreover, the result of an operation can depend on the concrete structure of the argument. Thus, two possibilities for defining functions exist: `Set` and `SetDelayed`. As a prerequisite for the following example, we introduce the commands `Expand` and `Factor`. We will use `Expand` in the following examples to show the difference.

`Expand[`*expression*`]`

multiplies out all products and (positive) integer powers appearing in the highest level of *expression*.

`Factor` does the opposite of `Expand`.

`Factor[`*expression*`]`

factors the highest level of *expression*, when possible.

Here is an example.

```
Expand[((1 + x)^2 + (2 + y)^3)^2]
```

Here is a univariate polynomial of degree 12 with the interesting property that its expanded square has fewer terms than the original polynomial [1✱], [15✱].

```
Expand[(1 + 2 x - 2 x^2 + 4 x^3 - 10 x^4 + 50 x^5 + 15 x^6 -
        220 x^7 + 220 x^8 - 440 x^9 + 1100 x^10 - 5500 x^11 -
        13750 x^12)^2]

Length[%]
```

Products lying at deeper levels are not immediately multiplied out. (We discuss in detail how we can reach them in Chapter 6.)

```
Expand[((1 + x)^2)^(1/2)]
```

The same statement for `Factor` in the equivalent expression.

```
Factor[Sqrt[x^2 + 2 x + 1]]
```

It also factors only the expression itself, not the parts of the expression.

```
Factor[x^2 + 2 x + 1]
```

`Factor` and `Expand` work only on polynomials. Other expressions, like trigonometric functions, can be expanded and factored using the specialized functions `TrigExpand` and `TrigFactor`.

`TrigFactor[`*expression*`]`

converts all powers of trigonometric functions in the highest level of *expression* into trigonometric functions with multiple angles.

Here are the powers of sin(x) and cos(y) rewritten as multiple angles.

```
TrigFactor[α Sin[x]^4 + β Cos[y]^6]
```

If we use `TrigFactor`, the powers of `Sin[y]` and `Cos[x]` are converted to `Sin[`*n* `x]`, `Cos[`*n* `x]`. But the resulting expression as a whole is not fully expanded.

```
TrigFactor[((1 + Sin[y])^2)^2 + ((2 + Cos[x])^3)^2]
```

Expanding the resulting expression yields a manifestly real result.

```
Expand[%]
```

Be aware that only explicitly occurring powers are always converted. If the powers are implicitly present (meaning only after expanding an expression), `TrigFactor` will frequently not transform the trigonometric functions.

```
TrigFactor[((2 + Sin[y])^2)^2] + TrigFactor[((3 + Cos[x])^3)^2]
```

In the next input, `TrigFactor` generates a result that is a product of trigonometric functions with argument $x/2$.

```
{TrigFactor[1 + Sin[x]], TrigFactor[1 + Cos[x]]}
```

`TrigFactor` operates also on hyperbolic functions.

```
TrigFactor[Expand[((1 + Sinh[y])^2)^2] + TrigFactor[((2 + Cosh[x])^3)^2]] //
        Expand
```

The trigonometric equivalent of `Expand` is `TrigExpand`.

`TrigExpand[`*expression*`]`

converts all trigonometric functions with multiple angles in the highest level of *expression* into powers of trigonometric functions.

Here are again two simple examples, one with trigonometric functions and one with hyperbolic functions.

```
TrigExpand[(2 + Cos[2 x] - 2 Cos[4 x] - 3 Cos[6 x])/32]

TrigExpand[1 + Cosh[3 x] + Tanh[5 x]]
```

We now explain how to define a function $f(x)$ depending on an arbitrary variable x that is to be specified later. The explanation is based on patterns standing for completely arbitrary expressions or whole classes of expressions. In *Mathematica*, these patterns are represented with `Blank` and `Pattern`.

`Blank[]`

or

`_`

is a pattern standing for an arbitrary *Mathematica* expression.

`Blank[`*head*`]`

or

`_`*head*

is a pattern standing for an arbitrary *Mathematica* expression with the head *head*.

`Pattern[`*x*`, Blank[]]`

or

much shorter *x*`_`

is a pattern named *x* standing for an arbitrary *Mathematica* expression.

`Pattern[`*x*`, Blank[`*head*`]]`

or

much shorter *x*`:_`*head*

or

still shorter *x*`_`*head*

is a pattern named *x* standing for an arbitrary *Mathematica* expression with the head *head*.

We look at the output of the short forms shown by `FullForm`.

```
FullForm[_]

FullForm[_Real]

FullForm[x_]

FullForm[x_Integer]
```

The colon in `Pattern` is typically not visible; however, it is needed in compound expressions.

Here, the colon is suppressed in the `InputForm`.

```
x:_h
```

```
FullForm[%]
```

```
InputForm[%%]
```

For patterns that do not contain `Blank`, the colon is needed. Here we input the (fixed) pattern `fixedPatternWith⋱ outBlank`.

```
InputForm[name:fixedPatternWithoutBlank]
```

The colon is also needed for the following compound expression.

```
theWholeExpression:(summand1_ + summand2_)
```

And the following (currently, semantically not very useful) expression does not use a colon or _; instead, the `Full⋱ Form` is used.

```
Pattern[pattern[1], Blank[value[1]]] // InputForm
```

Actually, such expressions using a colon would not be correct syntactically.

```
pattern[1]:Blank[value[1]]
```

Here is a more complicated expression using `Pattern` and `Blank`. Be aware that parentheses are needed for grouping. `a`, `b`, `c`, and `d` all represent the pattern `e`.

```
a:(b:(c:(d:e_))) // FullForm
```

Using `InputForm`, we also get the parentheses.

```
a:(b:(c:(d:e_))) // InputForm
```

For a function definition, the *x* in *x*_ must have the head `Symbol` (i.e., it cannot be a number, a product, or a composite expression).

Pattern structures of the form *x*:_*head* with `Blank` along with more general and specialized forms will be discussed in detail in Chapter 5. Now, we have everything we need to define our own function. Let us define a function that squares its argument and multiplies out the result. The following `x_` stands for an arbitrary *x* (remember the typeset convention to use italic slant for user-supplied arguments); it is only called `x` in this definition of our function, and it represents a pattern standing for one arbitrary expression.

```
multiplyItOut[x_] = Expand[x^2]
```

Here, we use `multiplyItOut` with various arguments (not the `x` from the above definition).

```
multiplyItOut[x]
```

```
multiplyItOut[ξ]
```

```
multiplyItOut[5]
```

```
multiplyItOut[i]
```

```
multiplyItOut[I]
```

No expansion happens in the following example.

```
multiplyItOut[2 + Sqrt[2] I]
```

In case the real or imaginary part are inexact numbers, the square autoevaluated to one complex number with approximate real and imaginary parts.

```
multiplyItOut[2 + Sqrt[2.] I]
```

```
multiplyItOut[2. + Sqrt[2] I]
```

The `x` in the first of the examples above has nothing to do with the `x_` on the left-hand side of `multiplyItOut`, or with the `x` on the right-hand side in the definition of `multiplyItOut`. The `x` on the right-hand side in a function definition is only defined locally for the sake of defining the function. This `x` relates only to the `x` on the left-hand side in the form `x_` (in case of *f*`[x_] =` *somethingContaining*`x`. The right-hand side of the definition is immediately evaluated, which means if `x` already has a value, this value is used). The `x`, `5`, `i`, and `I` we gave were actual arguments of the function `multiplyItOut`.

The following uses of `multiplyItOut` show the importance of the word *arbitrary* in the above discussion.

```
multiplyItOut["this is a string"]
```

```
multiplyItOut[Times]
```

```
multiplyItOut[hj[tz[ui[t]]]]
```

```
multiplyItOut[multiplyItOut[multiplyItOut[2]]]
```

```
multiplyItOut[garbage can]
```

```
multiplyItOut[Sqrt[2]]
```

Note that the `x_` in our definition of `multiplyItOut` stands for exactly one arbitrary argument, not for zero arguments, or for two or more arguments. If we use `multiplyItOut` without a variable or with more than one variable, it does not do anything, because now the pattern used in the definition of the function does not match.

```
multiplyItOut[]
```

```
multiplyItOut[2, 3]
```

```
multiplyItOut[2, 3, h]
```

The following argument `(1 + 2 x) (2 + 3 y)` is not expanded further because, at the time of the definition of the function `multiplyItOut`, we already multiplied and `Expand` is no longer present in the definition of `multi⋱ plyItOut`.

```
multiplyItOut[(1 + 2 x) (2 + 3 y)]
```

Using `??`, we see the current definition associated with `multiplyItOut`.

```
??multiplyItOut
```

We now define another function that squares its argument, and then multiplies the result out. In the following example, we use `:=` instead of = as above, which will make a big difference under certain circumstances.

```
multiplyItOutWithColon[x_] := Expand[x^2]
```

The next input gives the desired result.

```
multiplyItOutWithColon[(1 + 2 x) (2 + 3 y)]
```

```
multiplyItOut[(1 + 2 x) (2 + 3 y)]
```

And, of course, explicit complex numbers get multiplied out too.

```
multiplyItOutWithColon[2 + Sqrt[5] I]
```

Next, we define a function making use of the possibility of specifying the head of the argument.

```
fxo[x_fxo] = x;
```

The definition is applied as often as possible.

```
fxo[fxo[fxo[x]]]
```

If the argument does not have a head that is `fxo`, nothing is done.

```
fxo[fxo[fxo[x]]]
```

Here is another example, wherein the definition of the function is applied several times.

```
recursivelyApply[0] = 0;
recursivelyApply[x_Integer] := recursivelyApply[(x - 1)/2];
```

The computation of the example is accomplished by computing in order.

```
recursivelyApply[31]
```

The value of the last expression is `0` and follows directly from the above definition. Here is a sketch of the sequence of evaluations:

```
 recursivelyApply[31] ⟹ 7
  recursivelyApply[ 7] ⟹ 3
   recursivelyApply[ 3] ⟹ 1
    recursivelyApply[ 1] ⟹ 0
     recursivelyApply[ 0] ⟹ 0
```

The examples above explain the difference between using `=` and `:=`.

`Set[`*x*`,` *y*`]` or *x* `=` *y*

immediately evaluates *y* and assigns the result to *x*. From then on, whatever *y* evaluated to, this value of *y* will be substituted for every further appearance of *x*.

`SetDelayed[`*x*`,` *y*`]` or *x* `:=` *y*

assigns the unevaluated value of *y* to *x*. When *x* is evaluated later, the value of *y* at this time will be substituted for *x*.

The definition of functions for arbitrary arguments involves a pattern on the left-hand side.

f`[`*x*`_]` `=` *functionOfx*

defines a function *f* for which any arbitrary argument can be given for *x*. The computation of *functionOfx* is carried out to the extent possible when *f* is defined. If *y* is later input to *f*`[`*y*`]`, *y* will replace any instances of *x* in *functionOfx* and the resulting expression will be evaluated further, if possible.

f`[`*x*`_]` `:=` *functionOfx*

defines a function *f* for which any arbitrary argument can be given for *x*. The computation of *functionOfx* is not carried out until *f* is called with some particular argument *y*.

The `FullForm` of `f[x_] = functionOfx` is as follows: `Set[f[Pattern[x, Blank[]]], functionOfx]`

The `FullForm` of `f[x_] := functionOfxLater` is as follows:

`SetDelayed[f[Pattern[x, Blank[]]], functionOfxLater]`

Their `FullForm` cannot be seen by using the following construction.

```
FullForm[f[x_] = functionOfx]
```

```
FullForm[f[x_] := functionOfxLater]
```

In this construction, the argument of `FullForm`, that is, the function definition itself, is evaluated and then `FullForm` applies. We will discuss later in this chapter how to generate the above `FullForm` programmatically.

Note that the result of `Set` is the right-hand side of the input value, whereas the result of `SetDelayed` is `Null` (meaning "nothing"); that is, we only have a function definition, and no value has been returned. The right-hand side cannot be returned because it will not be evaluated.

`Integrate` is another command in which the difference between `Set` and `SetDelayed` is very important.

`Integrate[`*f*`, `*x*`]`
computes the indefinite integral of *f* with respect to the variable *x*.

Here is an example.

```
Integrate[x Sin[x], x]
```

We now define our own integration program, which we call `integrate`. As a first try, we define it using the following input.

```
integrate[fu_, x_] = Integrate[fu, x]
```

```
??integrate
```

This time, we implemented a two-argument function; both arguments were specified using the construction *var*_. The right-hand side of the definition of `integrate` was evaluated immediately, and at this time `fu` did not depend on `x`. The result of the integration is `fu*x.` (`fu` was considered to be an `x`-independent constant.)

From now on, `integrate` is associated with the function definition `integrate[fu_, x_] = fu*x`.

```
??integrate
```

```
integrate[x^3, x]
```

The following definition is what we probably want.

```
integrateNew[fu_, x_] := Integrate[fu, x]
```

```
integrateNew[x^3, x]
```

Note that in this example, the simplest solution would have been `integrate = Integrate`.

When in doubt, it is often better to use := instead of =. However, the price of always using := is possibly a substantial loss of efficiency, depending on the complexity of the right-hand side, because the operations defining it may have to be carried out more often than is necessary.

Note that with both `Set` and `SetDelayed`, variables on the right-hand side cannot be assigned any value if they also appear on the left-hand side inside `Pattern`. The right-hand side in the following example consists of two parts to be carried out for a given `xyz`. First, it is to be squared, and then the sin has to be taken.

```
fq1[xyz_] = (xyz = xyz^2; Sin[xyz])
```

The large number appearing in the last error message is a high power of 2. We will discuss the reason for this in the next chapter.

```
Log[2, %[[1, 2]]]
```

Here is the equivalent construction using `SetDelayed`.

```
fq2[xyz_] := (xyz = xyz^2; Sin[xyz])
fq2[1]
```

In both cases, we get an error message (covered in Chapter 4). Note that we get different error messages with `Set` and `SetDelayed`. In the case with `Set`, the recursive definition is carried out about 256 times, and then *Mathematica* stops this process. We will discuss the failure in the `SetDelayed` case in a moment.

Here is a definition using `SetDelayed`. It too leads to a recursion error.

```
(lhs : ff[x_]) := ℬ[lhs]
```

```
ff[3];
```

The whole left-hand side of the definition is named `lhs` in the pattern. When `ff` is called with an argument *arg*, it evaluates to `ℬ[ff[`*arg*`]]`, which again causes the `ff[`*arg*`]` to evaluate, and so on.

> Be aware of the following: After defining *f*`[`*x*`_] =` *something*(*x*) or *f*`[`*x*`_] :=` *something*(*x*), using *f*`[`*argument*`]` causes *every* occurrence of *x* in the right-hand side of the definition to be replaced by *argument.* This process may lead to unexpected results.

Here, this process is demonstrated.

```
noGo[x_] := (x = 11)
```

```
myNewVar = 1;
```

```
noGo[myNewVar]
```

`myNewVar` did *not* get the value 11 (although 11 was given as the output), because after substitution of `1` for `x` in the right-hand side of the definition of `noGo`, we had `Set[1, 11]`. This assignment is impossible to do.

```
myNewVar
```

```
1 = 11
```

For the same reason, the above `SetDelayed` construction, which had the variable `xyz` on the left- and the right-hand side, failed.

Note that *head* in `name_Blank[`*head*`]` cannot itself contain a `Blank`. We can, in principle, make the following definition.

```
h1[Pattern[x, Blank[Blank[h2]]]] := 2
```

```
??h1
```

However, our definition of `h1` does not match any head, as we might expect by analogy with the fact that *argument*_ matches any argument *argument*.

```
h1[h2[h3][x]]
```

It just matches the special head `Blank[h2]`.

```
h1[Blank[h2][xy]]
```

Of course, to have a function taking any argument of the form *arbitraryHead*`[x]`, we could define this or related constructions like `head_[argument_]` or `_[_]`.

```
extractHead[head_[x]] := head
```

Now the following example would work.

```
extractHead[testHead[x]]
```

In case *head*[*x*] does not evaluate, we could have also made the following definition.

```
extractHead2[head_[x_]] := head[x][[0]]
```

```
extractHead2[Sin[Pi/E]]
```

Functions can be defined not only with variable arguments, that is, with patterns that can stand for many potential arguments, but also for arbitrary special arguments and/or variable types. Such definitions are possible with both Set and SetDelayed.

Here is a somewhat exotic but, for our purposes, useful construction. We use a pattern with _, define the function for special values, and use one nested pattern. We define a function mySpecialFunction containing a different definition for each of the following cases.

```
(* four definitions that match classes of arguments *)
mySpecialFunction[x_Integer]       := x^2;
mySpecialFunction[x_Real]          := x^4;
mySpecialFunction[x_Rational]      := x^6;
mySpecialFunction[x_Complex]       := x^8;
(* four definitions for concrete arguments *)
mySpecialFunction[x]               := nowJustx;
mySpecialFunction[Infinity]        := nowInfinity;
mySpecialFunction["stringSpecial"] := nowASpecialString;
mySpecialFunction[3]               := specialValueFor3;
(* one definition for arguments with the head myHead *)
mySpecialFunction[_myHead]         := "WithMySpecialHead";
```

In the next input definition, a pattern appears inside of inside. inside is a fixed head, but the argument *x* is variable.

```
mySpecialFunction[inside[x_]] := withInsideFunction[x];
```

Here is the current definition of mySpecialFunction.

```
?? mySpecialFunction
```

This definition of mySpecialFunction always yields the correct value if applied. With an argument *x* of type Integer, we get the argument squared.

```
mySpecialFunction[2]
```

With the argument equal to 3, we get specialValueFor3.

```
mySpecialFunction[3]
```

With a rational argument, we get the sixth power of the argument.

```
mySpecialFunction[2/9]
```

When we input 9/3, it is not treated as a rational argument because it is first simplified to 3.

```
mySpecialFunction[9/3]
```

With a Real argument, we get the fourth power of the argument.

```
mySpecialFunction[2.]
```

If we input 2 + I 0, the argument is simplified to 2. Then the argument has the head Integer before mySpecial⋱ Function is evaluated, and we get 4.

```
mySpecialFunction[2 + I 0]
```

With an argument of type Complex, we get the eighth power of the argument.

```
mySpecialFunction[2. + I 0.0]
```

Here, again, the argument is simplified to type `Rational`, and we get `x^6`.

```
mySpecialFunction[2/3 + I 0]
```

Next, we input an argument of type `String`. We have not given a definition for an arbitrary element of this type. Thus, nothing is computed, and the result remains in the form *function* [*argument*].

```
mySpecialFunction["string"]
```

However, for the special argument `"stringSpecial"` of type `String`, we did give a nontrivial definition.

```
mySpecialFunction["stringSpecial"]
```

For the special variable `x`, we get `nowx`.

```
mySpecialFunction[x]
```

No definition was given for a general arbitrary variable without the head specification; so, if the input is of this type, nothing is computed.

```
mySpecialFunction[y]
```

Here is a look at the special structure `mySpecialFunction[inside[…]]` with an arbitrary inside argument.

```
mySpecialFunction[inside[arbitraryInsideArgument]]
```

The head of the actual argument inside `inside` does not matter.

```
mySpecialFunction[inside[3]]
```

When giving a definition of the form _*head*, only the head is important. It does not matter how many arguments are actually present.

```
mySpecialFunction[myHead[1, 2, 3, 4, 5, 6, 7, 8, 9]]

mySpecialFunction[myHead[]]
```

Note that in the definition of `mySpecialFunction`, we have only used `SetDelayed`. In defining a function *f*, it is possible to mix `Set` and `SetDelayed` definitions arbitrarily, so long as the left-hand sides differ. If they are equal, the last definition given applies. (This discussion supposes that the `Condition` command, which we will discuss in Chapter 5, is absent.)

> It is possible to give short implementations of complex functions using the structure `Blank[`*head*`]`, where the definition of the function depends on the type of its argument.

Here is an example in which `SetDelayed` has to be used. Every symbol *symbol* can be written as *symbol* `/ 1` and thus has the trivial denominator 1.

```
ABC[arg_Rational] = Denominator[arg];

ABC[5/6]

??ABC
```

This assignment happened the moment `Denominator[arg]` was calculated in the definition of `abc1`. The `1` was returned when `ABC[5/6]` was called.

Now, the denominator is extracted only for concretely prescribed arguments. (Note that the denominator of an approximative number is the integer `1`.)

```
abc[arg_Rational] := Denominator[arg];
abc[arg_Real] = arg;
{abc[5/6], abc[32.8]}
```

Function definitions with the structure *arg_* are also possible for functions with several arguments. We demonstrate this, with the following function `xyzxyz`.

```
xyzxyz[] := 222;
xyzxyz[_] := 333;
xyzxyz[x_] := 444;
xyzxyz[x_, y_] := 555;
xyzxyz[x_, y] := 666;
xyzxyz[x_, y_, z_] := 777;
```

Here is the result of `xyzxyz` called with a various numbers of arguments.

```
{xyzxyz,
 xyzxyz[x_],
 xyzxyz[],
 xyzxyz[hhh],
 xyzxyz[1, 1],
 xyzxyz[1, y],
 xyzxyz[1, 2, 3]}
```

It is not possible to simultaneously assign a value to a symbol used as a variable and to define a function with the same name.

```
ppo = 6;
ppo[x_] := x^2
```

In reverse order, no problem would occur in defining the function.

```
opp[x_] := x^2
opp = 6
```

But a call on the function will often not give a useful result.

```
opp[6]
```

```
opp[6] // N
```

Using `Set`, we have the same problem.

```
ppoSet = 6;
ppoSet[x_] = x^2
```

```
oppSet[x_] = x^2
oppSet = 6
```

```
oppSet[6]
```

Because a function can be defined to give different values for different types of arguments, many programming advantages exist. This feature was implemented in *Mathematica* on purpose. In complicated calculations, it may happen from time to time that the head of an expression is only determined during the course of a calculation, and that this current head has to be used to match the pattern in a function definition. Here is a function definition working only for real arguments.

```
uOnlyForRealArg[x_Real] := x^2;
```

If we call this function with 0 as an argument, it remains unevaluated because 0 has the head `Integer`.

```
uOnlyForRealArg[0]
```

Applying `N` to the last expression converts the integer 0 into the real number 0.0 and the definition above for `uOnly`·

ForRealArg matches.

```
uOnlyForRealArg[0.0]
```

For a complex approximate zero, the definition above does not fire.

```
uOnlyForRealArg[0.0 + 0.0 I]
```

The last shown behavior might be unexpected, but remember that the head of 0.0 + 0.0 I is Complex and not Real.

> When several contradictory definitions are given for a function, the more specific ones are used before the more general ones.

The typical structures from general to specific are f[x_] $\Longrightarrow f$[x_$head$] $\Longrightarrow f$[$xSpecial$]. For example, we first give a definition.

```
aFunction[r_] := 3r;
aFunction[2] := 2;
aFunction[i_Integer] := 2 i;
```

The rules have been reordered.

```
??aFunction
```

Here are two “equally specific” rules.

```
fpq1[p_, q] = 1;
fpq1[p, q_] = 2;
```

Here are the currently stored definitions for fpq1.

```
??fpq1
```

Then, if the first argument is p, or the second one is q, everything is unique. What happens in the case when the two arguments are just p and q? Which definition is more general?

```
{fpq1[p, 1], fpq1[1, q], fpq1[p, q]}
```

> When several definitions are given for a function that are of equal “generality”, or if *Mathematica* cannot tell which is more general, the definitions are used in the order in which they were input.

This fact means that the values of functions may depend explicitly on the order in which their definition is input. Suppose we define a function fpq2 in the same way as fpq1, but reverse the order of the input.

```
fpq2[p, q_] = 2;
fpq2[p_, q] = 1;
```

```
{fpq2[p, 1], fpq2[1, q], fpq2[p, q]}
```

We should emphasize once more that function definitions go immediately into effect, instead of later when the functions are applied to a nonpattern argument. This process happens even if they act inside other function definitions, because the arguments are evaluated and already-known definitions are used.

To illustrate this fact, we consider the following incorrect attempt to mimic the operation of the built-in function Expand. The first definition multiplies out an expression of the form a (b + c) to give a b + a c, and the second definition expresses the desire that a multiple application of the following firstExpandAttempt should be the same as one application. (In principle, this is superfluous and should happen by itself.) The third definition multiplies out every summand individually. (For the sake of simplicity, we do not attempt to program the evaluation of powers, etc.)

```
firstExpandAttempt[a_ (b_ + c_)] := a b + a c
```

```
firstExpandAttempt[firstExpandAttempt[a_ (b_ + c_)]] :=
                         firstExpandAttempt[a (b  + c)]
```

```
firstExpandAttempt[a_ + b_] := firstExpandAttempt[a] +
                               firstExpandAttempt[b]
```

With ?, we see that we have not defined what we wanted; and the argument `firstExpandAttempt[ a_ (b_ + c_)]` on the left-hand side of the second definition for `firstExpandAttempt` is computed according to the first function definition.

```
?firstExpandAttempt
```

Thus, the following example fails.

```
firstExpandAttempt[firstExpandAttempt[(a + b) (c + d)]]
```

Now, we change the order in which the definitions are input so that the equivalence of the repeated application of `secondExpandAttempt` is programmed first.

```
secondExpandAttempt[secondExpandAttempt[a_ (b_ + c_)]] :=
                           secondExpandAttempt[a (b + c)]
```

```
secondExpandAttempt[a_ (b_ + c_)] := a b + a c
```

```
secondExpandAttempt[a_ + b_] := secondExpandAttempt[a] +
                                secondExpandAttempt[b]
```

Now, we have exactly what we wanted.

```
?secondExpandAttempt
```

```
secondExpandAttempt[secondExpandAttempt[(a + b) (c + d)]]
```

The functions `Set` and `SetDelayed` introduced in this subsection are among the most important when working with *Mathematica*. Using `Set` or `SetDelayed`, *Mathematica* can, with enough available memory, work quickly with many thousand (or even million) rules associated with fixed functions.

Now that we have seen the importance of _ in *Mathematica*, we can understand why variable names of the form *name1_name2* are not possible.

```
one_plot
```

The last input does not define a symbol `one_plot`, but rather is a pattern named `one` with the head `plot`.

```
FullForm[%]
```

With several _, we get a product of three terms.

```
one_especially_beautiful_plot
```

```
FullForm[%]
```

Using parentheses appropriately, we get a another expression.

```
one_(especially_(beautiful_plot))
```

It is again a product.

```
FullForm[%]
```

But its detailed content is of not much value.

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

■ 3.1.2 Clearing Functions and Values

Sometimes, we want to remove values that have been assigned to functions or variables. This removal can be done using `Clear`.

> `Clear[`$symbol_1$`, `$symbol_2$`, … , `$symbol_n$`]`
>
> removes all values and definitions (numeric and symbolic) that have been assigned to the symbols $symbol_1$, $symbol_2$,… , $symbol_n$. Attributes are not removed.

Another possibility for removing values for symbols is `Unset`.

> `Unset[`*leftHandSide*`]` or *leftHandSide* `=.`
>
> removes any values assigned to *leftHandSide*.

If a new value is assigned to a quantity (not a function, which may have already been assigned something earlier), using either `Set` or `SetDelayed`, it has the new value.

```
(* use Set *)
κ = κκ;
κ = 11;
κ

(* use SetDelayed *)
ω := msdg;
ω := 11;
ω
```

For function definitions, the situation concerning `Set` and `SetDelayed` is somewhat different. If the left-hand sides of the assignment agree exactly, the old value is overwritten.

```
κ1[x_] := x^κ;
κ1[x_] = x^9;
κ1[2]
```

If the left-hand sides differ, for example, in the naming of the unimportant, dummy pattern variables, only the last definition is stored.

```
κ2[x_] := x^κ;
κ2[y_] = y^9;
??κ2

κ2[2]
```

In the following example, both definitions are active after identifying the two heads `myHead1` and `myHead2`. `myHead2` inside `y_myHead2` was not reevaluated after evaluating `myHead1 = myHead2`.

```
κ3[x_myHead1] := x^κ;
κ3[y_myHead2] = y^9;
myHead1 = myHead2;
??κ3
```

For removing definitions with identical left-hand sides, it does not matter if the definitions are done with `Set` or `SetDelayed`.

```
ϰ4[x_] := x;
ϰ4[y_] = y^2;
ϰ4[2]
```

`Unset` is a more precise tool than is `Clear`. Its use is recommended for the manipulation of definitions that consist of many parts. With `Clear`, we can only clear all definitions for a symbol (head `Symbol`, or input in `Clear` as a `String`).

```
Clear[f];
f[x_] := x^2;
f[1] = 1;
??f
```

```
Clear[f];
f[x_] := x^2;
f[1] = 1;
(* clear definition of the form f[x_] := ... *)
f[x_] =.
??f
```

Now that we know how to remove values assigned to variables, we return to the question of local variables in function definitions. Variables used as pattern names that appear on the left- and right-hand sides of a function definition have no effect outside the definition. If variables are used that have already been assigned values, these assigned values may affect the definition.

In the following example of `Set`, the right-hand side of the function definition is immediately computed. At this point, `x` has the value `assigned`, and the definition of `testFunction` will be stored as `testFunction[x_] = assigned`. Thus, the value of the function is actually independent of its argument.

```
x = assigned;
testFunction[x_] = x;
```

```
??testFunction
```

```
testFunction[x]
```

If a value is assigned to `x` only after the definition of `testFunction`, this leads to a different definition for `test⸌Function`; namely, `testFunction[x_] = x`, where the `x` on the right-hand side is now associated with the `x_` on the left-hand side.

```
Clear[x];
testFunction[x_] = x;
x = assigned;
??testFunction
```

Calling `testFunction` now with the argument `x` (which has the value `assigned`), according to the definition of `testFunction`, evaluates to `assigned`.

```
testFunction[x]
```

With `SetDelayed`, the right-hand side is computed only after the function is called. At this point and already at the time of making the definition in the following example, `x` has the value `assigned`.

```
Clear[x, testFunction];
x = assigned;
testFunction[x_] := x;
??testFunction
```

```
testFunction[x]
```

If we clear the value of `x` before the computation of the right-hand side, we get the current value of `x`, and because it has no assigned value, we just get `x`.

```
Clear[x]
```

```
testFunction[x]
```

A symbol can be completely removed with the function `Remove`.

`Remove[`*symbol*$_1$`,` *symbol*$_2$`, … ,` *symbol*$_n$`]`

removes the symbols *symbol*$_1$, *symbol*$_2$, … , *symbol*$_n$ along with their numeric and symbolic values, and any attributes assigned to them.

To illustrate, we define a function `fg` of two variables.

```
fg[x_, y_] := x y
```

With the arguments ξ and η, we get the following.

```
fg[ξ, η]
```

To find out what information is associated with the symbol `fg`, we use `??`.

```
??fg
```

We now cancel this definition.

```
Clear[fg]
```

The definition is gone, but the symbol `fg` itself is still available.

```
??fg
```

(We will come back to the meaning of `` Global` `` in Chapter 4.) To get rid of the symbol `fg`, we use the function `Remove`.

```
Remove[fg]
```

Now `??` gives a different result.

```
??fg
```

What happens if a symbol is removed using `Remove`, but it appears in other functions that have not been removed? So let us enter the following definitions.

```
storage = toSave[a, b, c]
```

```
Remove[a, b, c]
```

What is now in `storage`?

```
storage
```

```
InputForm[%]
```

The symbol `Removed` has the following meaning.

`Removed["`*symbol*`"]`

identifies all symbols that were variables that have been removed using `Remove`.

The reintroduction of the symbol `a` has no affect on the contents (arguments) of `storage`.

```
a
```

```
storage
```

Once something (a function, a variable) has been removed, the only way to recreate it is to enter the definition or its value again.

Often, we want to cancel a whole class of symbols. This cancellation can be done using strings as arguments of `Clear` and `Remove`.

`Clear[`*string*$_1$`,` *string*$_2$`, …,` *string*$_n$`]`

clears all numeric and symbolic values and definitions of objects that are matched by the strings *string*$_1$ or *string*$_2$ or … or *string*$_n$.

`Remove[`*string*$_1$`,` *string*$_2$`, … ,` *string*$_n$`]`

removes all numeric and symbolic values and definitions, along with the symbols themselves, of those objects represented by the strings *string*$_1$ or *string*$_2$ or …or *string*$_n$.

Here, we should mention that the string metacharacters (wild cards) `*` and `@` could be used. Remember, a string has to be enclosed in double quotes `"`*characters*`"`.

`*`

is used for any number, including none, of arbitrary characters.

`@`

is used for any number, including none, of arbitrary characters, excluding capital letters and `$`.

These metacharacters can also be used in other functions that make use of strings, for example, in `?`. Here, `?? @` typically gives a list of all user-defined symbols that are global in the current session (as they will generally not start with a capital letter).

```
?? @
```

Here are three assignments to symbols that all begin with `f`.

```
f1 = 1;
f2 = 2;
f3 = 3;
{f1, f2, f3}
```

Next, we clear the definitions of all functions beginning with `f`.

```
Clear["f*"];
{f1, f2, f3}
```

When symbols have been assigned values in a *Mathematica* session, we should not try to clear their values using `Remove["*"]` (and `Remove["@*"]` is dangerous because `"@"` matches the `$` character). Such an input will lead to a lot of error messages generated by attempts to clear built-in functions (see Section 3.2.2). Moreover, some important built-in functions will be lost. (In Chapters 4 and 6, we discuss how user-defined functions can be separated from all built-in functions.) Here is a list of all functions that would be removed.

```
wouldBeRemovedFunctions =
Select[Names["System`*"], ((FreeQ[#, Locked] &&
             FreeQ[#, Protected])&[Attributes[#]])&];

Short[wouldBeRemovedFunctions, 4]
```

```
Length[wouldBeRemovedFunctions]
```

To conclude this subsection, we now look at the following (somewhat) exotic construction.

```
f[x_] := (Remove[f]; x^2)
```

```
f[2]
```

```
f[2]
```

What happened in the second call of `f[2]`? `f[2]` remained unevaluated because in the first call of `f[2]`, the `f` itself (and its definition) was removed and the result is just `f[2]`.

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

■ 3.1.3 Applying Functions

Several different syntactical possibilities exist for applying a function of one variable to its single argument. The primary reason for this variety is to make programs easier to read and to emphasize and de-emphasize certain programming constructs. Here are the possibilities (we are already familiar with these for `N`).

Form	Name	Nesting
f[x]	standard form	f_1 [f_2 [x]]
f @ x	prefix form	f_1 @ (f_2 @ x)
x // f	postfix form	(x // f_2) // f_1

Here are the three possible ways of computing sin($\pi/4$) or, more precisely, of applying the function `Sin` to the argument `Pi/4`.

```
Sin[Pi/4]
```

```
Sin @ (Pi/4)
```

```
Pi/4 // Sin
```

Using the prefix form, the explicit use of brackets is important; `Sin @ Pi/4` is parsed as `(Sin @ Pi)/4`.

```
Sin @ Pi/4
```

For functions with two or more variables, two ways to apply a function exist: standard form and infix form.

Form	Name
f[x_1, x_2, … , x_n]	standard form
x_1 ~f ~ x_2 ~f ~ ⋯ ~ f ~ x_n	infix form

`Plus` is a typical example of a function with several variables.

```
Plus[1, 2, 3, 4]
```

```
1 ~ Plus ~ 2 ~ Plus ~ 3 ~ Plus ~ 4
```

`Set` also has two arguments.

```
a ~ Set ~ 2
```

```
??a
```

But we will rarely use `Set` in this form.

Be careful with your own functions for which no rules have been declared; in particular, the use of parentheses can produce results different from the expected ones. The infix form groups from the left.

```
1 ~ sulP ~ 2 ~ sulP ~ 3 ~ sulP ~ 4
```

But in infix form, no parentheses are added.

```
Infix[sulP[sulP[sulP[1, 2], 3], 4]]

Infix[sulp[1, 2, 3, 4]]
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

3.2 Options and Defaults

Many functions in *Mathematica* allow the user to make a number of choices; these include accuracy level, method (e.g., numerical integration or summation), colors, light sources, surface properties, line widths, labels for graphics, etc. Choices are realized in *Mathematica* by setting options. Options change details of the calculation process, and maybe the “value” of a result, but they do not change the “form” (shape) of the result, this means, for instance, that a function that returns a list in case no options are explicitly specified will also return a list in case any of its options are set to any possible value. If an option is not explicitly set, an appropriate default value will be used. We have already encountered the setting of options, such as `Heads -> True` in `Level`. Here is how they appear in *Mathematica*.

> Options are specified by *optionName* `->` *optionSetting*.

We will return to the exact structure of options (meaning the `FullForm` effect of `->`) in Chapter 5. Before discussing further the setting of options, we introduce a *very* useful function: the list `List` (table, vector, matrix, tensor, etc.). `List` is *the* typical *Mathematica* container for storing and collecting data (which is covered in detail in Chapter 6). We have already made some use of lists and will use them frequently later. `Level` for instance returned its result in form of a `List`. Now, we introduce them “officially”.

`List[`$expression_1$`, `$expression_2$`, … , `$expression_n$`]`

or

`{`$expression_1$`, `$expression_2$`, …, `$expression_n$`}`

is a list (sequence, ordered collection) of the expressions $expression_1$, $expression_2$, …, $expression_n$.

The internal representation is `List`, and the input is usually accomplished in the form `{… }`.

```
FullForm[{1, 2, 3, 4, 5, 6, 7, 8, 9}]

TreeForm[{1, 2, 3, 4, 5, 6, 7, 8, 9}]
```

The *i*th element of this list, which can be extracted using `Part`, is *i*, as we expect. (We use the input form of `Part`, *expr*`[[`*integer*`]]`.)

```
{1, 2, 3, 4, 5, 6, 7, 8, 9}[[4]]
```

Now, we return to our discussion of options.

`Options[`*symbol*`]`

gives a list of all possible options and their defaults for the symbol (function) *symbol*. Here, *symbol* is typically one of the functions in the system, or in some package.

`Options[`*expression*`]`

gives a list of the values of all options and their current settings for *expression*. Options with the default `Automatic` are also included. Here, *expression* is typically an expression created by the user with the help of system commands with options.

`Options[`*expression*`, ` *optionName*`]`

gives the current value of the option *optionName* in *expression*. Options whose values have been set with the default `Automatic` are not included. Here, *expression* is typically an expression created by the user with the help of system commands with options.

An excellent example of a *Mathematica* function with options is `Plot`. Among the functions with many options (we show the 15 leading functions and how many options they have next), it is the simplest one.

```
Take[Sort[{ToString[#], Length[Options[#]]}& /@ (* the functions *)
          (ToExpression[#, InputForm, Unevaluated]& /@ Names["*"]),
          Last[#1] > Last[#2]&], 15] // (* show as table *)
               TableForm[#, TableAlignments -> {Left}]&
```

The functions `Notebook`, `Cell`, and `StyleBox` used in *Mathematica* notebooks have the most options. From the kernel functions, the 3D and 2D plotting and graphics functions have the most options.

`Plot[`*function*`(`*x*`), {`*x*`, ` x_0`, ` x_1`}, ` *options*`]`

draws the graph of the function *function*(x) in the interval $x_0 \leq x \leq x_1$ using the options *options*.

Here is an example in which no options have been explicitly set. `Plot` returns a `Graphics`-object and as a "side effect" generates a "picture".

```
plot0 = Plot[x^(Sin[x]^2), {x, Pi, 5 Pi}]
```

Now, we want to change a few options to remove the axes, increase the number of sample points, draw the curve with a thicker line, change the height-width ratio, and draw a box around the plot.

```
plot1 = Plot[x^(Sin[x]^2), {x, Pi, 5 Pi},
             (* options for different-looking plot *)
             AspectRatio -> 1/3, PlotPoints -> 250,
             Frame -> True, PlotRange -> All, FrameTicks -> None,
             PlotStyle -> {Thickness[0.016], RGBColor[0, 0, 1]}]
```

Here is a list of all options of `Plot`. (We discuss their influence on the plot and their possible settings in great detail in Chapter 1 of the Graphics volume [62✱].)

```
Options[Plot]
```

```
Length[%]
```

Because we did not set any options via *optionName* `->` *optionValue* in `plot0`, `Options[plot0]` and `Options[` `Plot]` are essentially the same. The only differences are options that have already been used, and are no longer changeable. Such options are `PlotPoints`, `MaxBend`, `PlotDivision`, and `PlotStyle`. Once a graphic is produced, these options cannot be changed any more and so do not appear in the following list. Be aware: In comparison to the built-in function `Plot`, `plot0` is a graphic or, to be more accurate, a `Graphics`-object.

```
Options[plot0]
```

In `plot1`, `AspectRatio` and `Frame` are also different. Here, we filter out the options that are different.

```
Complement[Options[plot1], Options[plot0]]
```

Some functions, especially numeric functions that carry out complicated algorithms allow the specification of options specific to a certain algorithm. In such cases, options can occur in a nested manner such as `Method -> {`*concrete Method*`, {`*optionOfConcreteMethod*$_1$`->` *value*$_1$`, ...}}`. Here is an example. The function `NMinimize[{`*function*`, `*constraints*`}, `*vars*`, `*options*`]` tries to find the global minimum of the function *function* domain of variables `vars` obeying the constraints *constraints*. One possible method option is to carry out a random search over the domain. The number of search points to be used before other techniques are applied is specified with the suboption `"SearchPoints"`. Using a larger number of search points will frequently result in a smaller returned minimum. Here are two examples.

```
NMinimize[{Cos[x/y] Cos[10 y/x], x^2 + y^2 < 1} , {x, y},
          Method -> {"RandomSearch",
                     (* suboptions for the "RandomSearch" method option *)
                     {"SearchPoints" -> 10}}]

NMinimize[{Cos[x/y] Cos[10 y/x], x^2 + y^2 < 1} , {x, y},
          Method -> {"RandomSearch", {"SearchPoints" -> 1000}}]
```

Similarly, nested option can sometimes be used for built-in functions that call other functions (or itself recursively). In such cases, the nested options can be used to set the options of these other functions.

Still more information can be obtained with `AbsoluteOptions`.

`AbsoluteOptions[`*expression*`]`

gives a list of the values of all options of *expression*. Here, *expression* is typically an expression created by the user using system function with options. The values of the options with the default `Automatic` are also listed.

`AbsoluteOptions[`*expression*`, `*optionName*`]`

lists the specified value of the option *optionName* in *expression*. Here, *expression* is typically an existing function in the system or from a package.

Here, we look at all the options of `plot1`. Now, `PlotRange`, `FrameLabel`, and so on, have different values. (To avoid a very long output, we use the postfix function `(# /. (Ticks -> ticks_) :> (Ticks -> Short[ticks, 4]))&` to shorten the right-hand side value of the `Ticks` option.)

```
AbsoluteOptions[plot1]  // (* abbreviate tick specifications *)
                (# /. (Ticks -> ticks_) :> (Ticks -> Short[ticks, 4]))&
```

Using *optionName* `->` *specialEffect*, we can change the value of the option *optionName* inside a function. Frequently, we do not want to type this in repeatedly. We could change the options for a command with `Options[`*function*`] =` *listOfTheOptionsAndTheSettings*, but this could be mean to type this repeatedly for each call of a function. It is easier to use `SetOptions`.

`SetOptions[`*symbol*`, `*option*$_1$ `->` *specificValue*$_1$`, `*option*$_2$ `->` *specificValue*$_2$`, … ]`

sets the options *option*$_i$ of the symbol *symbol* to *specificValue*$_i$ for all *i*.

Without explicitly setting a value for the option `PlotPoints` of `Plot3D`, a three-dimensional (3D) plot of a function uses exactly 15 sample points in each dimension. `Plot3D` returns a `SurfaceGraphics`-object. We will discuss it in detail in Chapter 2 of the Graphics volume [62✱].

```
Plot3D[1/(2 + Sin[x y]), {x, -Pi, Pi}, {y, -Pi, Pi}]
```

In plotting repeatedly functions that are quite oscillating, we may want to alter the global default value for `Plot`

`Points`. Here, we use `SetOptions` to change the value of `PlotPoints` for all succeeding uses of `Plot3D`. `SetOptions` gives a list of all the options and their current settings.

```
SetOptions[Plot3D, PlotPoints -> 35]
```

Here, we plot the same function, but 35 points are used, as specified in the last input.

```
Plot3D[1/(2 + Sin[x y]), {x, -Pi, Pi}, {y, -Pi, Pi}]
```

The output of `Plot3D` is a list, with the most recent valid options and settings of `Plot3D`, that usually causes a graphical image to be displayed on the screen (as a "side effect").

At this point, we should mention that not all options of all built-in functions are always user-settable through the options of the function directly. Some less frequently used options are set through the system options `Developer`⋱ SystemOptions[]`. We will occasionally make use of the possibility to influence the behavior of functions through system options.

The following input returns the system options that influence single functions or groups of functions corresponding to the system option names.

```
Select[First /@ Developer`SystemOptions[],
       StringMatchQ[#, "*Options"]&]
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

3.3 Attributes of Functions

Functions have a variety of properties from the standpoint of mathematical analysis. For example, they may be commutative, associative, etc. *Mathematica* also deals with such properties, along with others that are more specific to computer algebra. They can be directly associated with the corresponding symbol (meaning the function name). The attributes associated with a symbol can be obtained by using `Attributes`.

> `Attributes[`*symbol*`]`
>
> gives a list of the attributes that *symbol* carries.

Here are some examples.

```
Attributes[Plus]

Attributes[Times]

Attributes[Position]

Attributes[Sin]
```

`Orderless` is the *Mathematica* analog of commutativity.

> `Orderless`
>
> is an attribute of a function with two or more variables, and it indicates that the variables should automatically be put in their canonical order.

The sum of all letters, input in reverse alphabetical order, will automatically be reordered by an `Orderless` function, such as `Plus`.

```
z + y + x + w + v + u + t + s + r + q + p + o + n +
m + l + k + j + i + h + g + f + e + d + c + b + a
```

The following products of indexed quantities is ordered to give sorted “indices”.

```
a5 * a4 * a3 * a2 * a1 * a0
```

```
a[5] * a[4] * a[3] * a[2] * a[1] * a[0]
```

Attributes can be assigned by the user to both system- and user-defined functions. This process is done with `SetAttributes`.

`SetAttributes[`*symbol*`, ` *attributes*`]`
adds the attribute *attributes* to the list of attributes of *symbol*.

Attributes should be set before any other definition or value assignment.

For some definitions, attributes can also be set later, to avoid the use of the property expressed through the attribute. We now define a commutative function `commutativeFunction`, which automatically puts its arguments into canonical order.

```
SetAttributes[commutativeFunction, Orderless]
```

```
commutativeFunction[y, x]
```

```
Attributes[commutativeFunction]
```

Note that numbers are ordered lexicographically even though we have not given any explicit function definition.

```
commutativeFunction[12, 11]
```

Note also that the attributes (`Orderless` as well as other attributes) do not change anything in the lowest level of the arguments if the head is a composite function.

```
SetAttributes[cmf, Orderless]
```

```
cmf[2, 1][4, 3]
```

It is not possible to give composite heads attributes; the heads must be symbols.

```
SetAttributes[compHead[1, 2], Orderless]
```

`Flat` is the *Mathematica* analog for associativity.

`Flat`
is an attribute of a function with several variables causing
$f(f(a, b), c) = f(a, f(b, c)) = f(a, b, c)$ to be automatically applied.

We now define an associative function `associativeFunction`.

```
SetAttributes[associativeFunction, Flat]
```

An associative function or operation need not be commutative (matrix multiplication of square matrices is a typical example), and thus, `Flat` and `Orderless` have to be strictly distinguished. In `associativeFunction[c, b, a]`, the arguments are not reordered.

```
associativeFunction[c, b, a]
```

In the next examples, the `Flat` attribute has an effect: The result is not nested.

```
associativeFunction[a, associativeFunction[b, c]]
```

```
associativeFunction[associativeFunction[a, b], c]
```

In the process of evaluation, the properties originating from attributes are used as often as possible because *Mathematica*'s evaluation procedure is applied as often as possible (see Chapter 4).

```
associativeFunction[c,
        associativeFunction[a,
                associativeFunction[b1,
                          associativeFunction[b21, bb22]]]]
```

In particular, the following attribute is useful in simplifications using `Flat` (discussed in detail in Chapter 5).

`OneIdentity`

is an attribute of a function representing the property $x = f(x) = f(f(x)) = f(f(f(x))) = \ldots$ for the purposes of pattern matching (see Chapter 5).

Large lists of numbers or symbols are often built up during calculations. To apply functions automatically to every element, the functions must carry the `Listable` attribute.

`Listable`

is an attribute of a function causing the automatic application of the property $f[\{a_1,\ a_2,\ \ldots,\ a_n\}] \longrightarrow \{f[a_1],\ f[a_2],\ \ldots\ ,\ f[a_n]\}$.

`Sin` has the `Listable` attribute, as do all other built-in mathematical numerical functions.

```
Attributes[Sin]
```

Thus, we get the following result.

```
Sin[{Pi, Pi/2, Pi/3, Pi/4, Pi/5, Pi/6}]
```

We now define our own `Listable` function of three variables. The attribute `Listable` holds for all arguments, and it is applied to the three arguments in parallel.

```
SetAttributes[ourTripleSin, Listable];
ourTripleSin[{a, b, c}]
```

In cases with more arguments, corresponding ones are used together as arguments.

```
ourTripleSin[{a, b, c}, {1, 2, 3}, {x, y, z}]
```

In the next example, only the first argument is a list.

```
ourTripleSin[{a, b, c}, 123, xyz]
```

If the list arguments are of unequal length, an error message is generated.

```
ourTripleSin[{a}, {1, 2}]
```

One extremely important attribute for numeric evaluations is `NumericFunction`.

`NumericFunction`

is an attribute of a function that for numeric arguments represents a numeric quantity.

Here is a list of all built-in *Mathematica* functions that have the attribute `NumericFunction`.

```
Select[Names["*"], MemberQ[Attributes[#], NumericFunction]&]
Length[%]
```

The `NumericFunction` attribute can be set also for user-defined functions.

```
SetAttributes[myNumericFunction, NumericFunction];
```

Now, *Mathematica* considers every expression of the form `myNumericFunction[`*numericArgument*`]` as a numeric quantity. (The function `NumericQ` tests if an expression is a numeric quantity—see Chapter 5.) It is the user's responsibility to give appropriate definitions for `myNumericFunction` that are semantically sensible.

```
NumericQ[myNumericFunction[Pi]]
```

With the `NumericFunction` attribute, the function `myNumericFunction` evaluates nontrivially for the argument `Indeterminate`.

```
myNumericFunction[Indeterminate]
```

`Constant` is an important attribute for doing calculus.

`Constant`

is an attribute of a symbol ensuring that this symbol will identically vanish if a derivative, with respect to any variable, is applied.

To make use this attribute, we must be able to differentiate.

`D[`*function*`, {`x_1`, `i_1`}, {`x_2`, `i_2`}, …, {`x_n`, `i_n`}]`

gives $\frac{\partial^{i_1}\partial^{i_2}\cdots\partial^{i_n} function(x_1,x_2,\ldots,x_n)}{\partial x_1^{i_1}\,\partial x_2^{i_2}\cdots x_n^{i_n}}$, the i_1th partial derivative with respect to x_1, the i_2th partial derivative with respect to x_2,… , the i_nth partial derivative with respect to x_n of *function*. The possibility to interchange the order of the derivatives is automatically assumed (i.e., it is assumed that the Lemma of Schwartz holds and all occurring functions are smooth enough). When there is just one dependent variable, this can be written in a shorter form.

`D[`*function*`, {`*x*`, `*i*`}]` or *function*$\underbrace{\texttt{''}\cdots\texttt{''}}_{n \text{ times}}$`[`*x*`]`

gives the *i*th derivative of function with respect to *x*. For *i* = 1, we can write `D[`*function*`, `*x*`]` instead of `D[`*function*`, {`*x*`, 1}]`.

We illustrate the usage of the command `D` by computing a derivative of the following simple function of four variables.

```
multiArgumentFunction[w_, x_, y_, z_] = Cos[w^2] Exp[x] Log[y] z^2
```

Here is one of its higher derivatives.

```
D[multiArgumentFunction[w, x, y, z], {x, 1}, {y, 1}, {z, 2}]
```

Sometimes the function cannot be explicitly differentiated, which may be either because it is not explicitly defined, or because *Mathematica* does not know a rule for the differentiation; or if no such rule exists. (This is the case for some special functions with respect to some of their parameters, e.g., the Theta functions in Chapter 3 of the Symbolics volume [64✱].) When functions cannot be differentiated, an expression of the following form is returned. The integer appearing in the *i*th position inside the parentheses in a superscript describes how many times to differentiate with respect to the *i*th variable.

```
D[notExplicitlyDefinedFunction[x, y, z], (* differentiate 13 times *)
  {x, 2}, {y, 3}, {z, 4}, {x, 3}, {y, 1}]
```

The internal form of this object is somewhat complicated. (We cover this point in Chapter 1 of the Symbolics volume [64✱].)

```
FullForm[%]
```

Here is a rational function containing two arbitrary functions $p(x)$ and $q(y)$ and four constants *a0*, *a1*, *a2*, and *a3* [59✱], [39✱], [60✱].

```
u[x_, y_] = -D[q[y], y] D[p[x], x]/
              (a0 + a1 p[x] + a2 q[y] + a3 p[x] q[y])^2;
```

`pde` is a function representing a nonlinear partial differential equation in u with respect to x and y.

```
pde[u_, {x_, y_}] :=
2 (a0 a3 - a1 a2) u^3 + D[u, x] D[u, y] - u D[u, x, y]
```

The following input shows that, for all $p(x)$ and $q(y)$, the function $u(x, y)$ is a solution of the differential equation `pde`.

```
pde[u[x, y], {x, y}] // Factor
```

We return now to our discussion of the attribute `Constant`. The following differentiation leads to the expected result.

```
D[constantFunction[x, y], {x, 2}, {y, 2}]
```

We can declare `constantFunction` to be a constant with respect to differentiation.

```
Remove[constantFunction];

SetAttributes[constantFunction, Constant]
```

Then, although `x` and `y` are explicitly available as arguments, we get the following.

```
D[constantFunction[x, y], {x, 2}, {y, 2}]
```

The following “constants” in *Mathematica* have the attribute `Constant` (among them is our `constantFunction`.) (They are mostly mathematical constants. (The symbol `MachinePrecision` shares the property that after applying `N` it becomes a number with the mathematical constant.) While this does not imply that *constant*(x) is independent of x, such a use of *constant* is not recommended.)

```
Select[Names["*"], MemberQ[Attributes[#], Constant]&]
```

A class of attributes exists for dictating how rules may be added to built-in functions. Attempting to change a system function directly fails.

```
Sin[z] = siSiSinSinus[z]
```

This failure is because of the attribute `Protected`.

`Protected`

is an attribute of a symbol preventing its definition, or its values, from being changed. The attributes of symbols can be changed, however, even if the symbol carries the attribute `Pro`⋱
`tected`.

However, still tighter restrictions than `Protected` still exist. For example, `I` has the following property.

```
Attributes[I]
```

`Locked`

is an attribute of a symbol preventing its definition, values, and attributes from ever being changed.

If a symbol has certain attributes (but not the attribute `Locked`), it is also easy to remove them. This removal is done with `ClearAttributes`.

`ClearAttributes[`*symbol*`,` *attributes*`]`

removes the attributes *attributes* from the list of attributes of *symbol*.

```
Attributes[constantFunction]

ClearAttributes[constantFunction, Constant]
```

Now, the list of attributes of `constantFunction` is empty.

```
Attributes[constantFunction]
```

It is possible to change system functions because the attribute `Protected` can be removed.

```
Attributes[Sin]

ClearAttributes[Sin, Protected]

Attributes[Sin]
```

This new rule will now be applied everywhere.

```
Sin[z] = siSiSinSinus[z]

Sin[z]

Integrate[Cos[z], z]
```

Instead of using `ClearAttributes` to change system functions, we can use `Unprotect`. This allows the actual definitions of the functions to be changed.

`Unprotect[`*symbol*`]`

removes the attribute `Protected` from the list of attributes of *symbol*.

`Protect[`*symbol*`]`

adds the attribute `Protected` to the list of attributes of *symbol*.

Calling `Unprotect` or `Protect` gives the function unprotected or protected outputs. Using these functions, we modify the built-in function `Cos`.

```
Attributes[Cos]

Unprotect[Cos]

Attributes[Cos]

Cos[z] = coCoCosCosinus[z]

Protect[Cos]

Attributes[Cos]

Cos[z]
```

For later possible applications of `Sin` and `Cos`, we remove our modifications to them.

```
Unprotect[Sin, Cos];
Clear[Sin, Cos];
Protect[Sin, Cos];
{Sin[z], Cos[z]}
```

Note a function `Unlock` does not exist, so locked symbols cannot be changed in any way. Users can lock function, but not unlock them.

Sometimes, we want to prevent an expression from being immediately evaluated, for example, when we are interested in the structure of an expression rather than the result. This "freeze" is possible with the function `Hold`, which is not an attribute, but its most important property is caused by its attribute.

`Hold[`*expression*`]`

prevents the evaluation of *expression*.

Here is a sum computed to be 45 as soon as it is input or used in the argument of a function.

```
1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9
```

With `Hold`, it remains in its original form.

```
Hold[1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9]
```

If something is enclosed in `Hold`, no reordering takes place.

```
Hold[4 + 3 + 2 + 1]
```

With `Hold`, we can now take care of several things that were left unsettled in Chapter 2.

```
FullForm[Hold[Divide[a, b]]]
```

However, `/` is not identical to `Divide`.

```
FullForm[Hold[a/b]]
```

And `-` is not identical to `Subtract`.

```
FullForm[Hold[a - b]]
```

The output can be kept in its original form without the explicit visible `Hold` by using `HoldForm`.

`HoldForm[`*expression*`]`

prevents the immediate evaluation of *expression* and displays *expression* without `Hold` in `OutputForm` and `StandardForm`.

```
HoldForm[1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9]
```

Although this output does not show it, the `HoldForm` is still there, as can be seen with `FullForm`.

```
FullForm[%]
```

`Hold` and `HoldForm` have the following attributes.

```
Attributes[Hold]
```

```
Attributes[HoldForm]
```

One function related to `Part`, which makes use of `Hold`, is `HeldPart`.

`HeldPart[`*expression*`, `*position*`]`

takes the part specified by *position* and wraps it before any further evaluation in `Hold`.

So, we can extract `1 - 1` from the following expression without it resulting in a `0`.

```
HeldPart[Hold[Sin[1 - 1]], 1, 1]
```

If we want to pass an expression to another function without evaluating it, we can use `Unevaluated`. (It is also not an attribute.)

`Unevaluated[`*expression*`]`

prevents the immediate evaluation of *expression*, but gives *expression* immediately as an argument to a function, without evaluating *expression* first if used as an argument in a function.

`Unevaluated` has the following attributes. We will discuss the attribute `HoldAllComplete` in a moment.

```
Attributes[Unevaluated]
```

Here is the sum from above used as an argument of `Unevaluated`.

```
Unevaluated[1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9]
```

The expression `1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9` has nine terms. The direct approach to determining the number of summands in this simple sum will not work.

```
Length[1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9]
```

Here is an argument of `Length`, using `Unevaluated`, that causes `Length` to measure the length of the unevaluated sum of the nine terms.

```
Length[Unevaluated[1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9]]
```

Because `Hold[`*argument*`]` has length 1, independent of the concrete structure of *argument*, here is what we get with `Hold` instead of `Unevaluated`.

```
Length[Hold[1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9]]
```

Most built-in functions without `Hold`-like attributes (see below) evaluate their arguments, but some do not. A notable exception is `Times`. As a result, we get the following unexpected result 0 (instead of a message and the result `Indeterminate`).

```
Times[0, Unevaluated[Infinity]]
```

It is also possible to use the same attributes making `Hold` and `Unevaluated` work to endow other functions with appropriate attributes to prevent their immediate evaluation. The "magical" attributes avoiding evaluation are `HoldAll`, `HoldFirst`, `HoldRest`, and `HoldAllComplete`.

`HoldAll`

is an attribute preventing the immediate evaluation of all arguments of a function.

`HoldFirst`

is an attribute preventing the immediate evaluation of the first argument of a function.

`HoldRest`

is an attribute preventing the immediate evaluation of all but the first argument.

Here is an example.

```
SetAttributes[holdFunction, HoldAll]
```

```
holdFunction[3 4, 5 + 8]
```

With the attribute `HoldAll`, we can nicely demonstrate the scope of activity of the attribute attached to a function.

```
SetAttributes[hff, HoldAll]
```

Only the “direct” argument of `hff` is not evaluated. The following arguments are not in the scope of activity.

```
hff[1 + 1][1 + 1]
```

But, on the other hand, be aware that attributes can be given only to symbols, not to constructions like `hff1[1 + 1]`.

```
SetAttributes[hff1[1 + 1], HoldAll]
```

From time to time, the attributes `HoldAll`, `HoldFirst`, and `HoldRest` will be used for user-defined functions, especially when it is necessary to scope variables. They also play a very important role in the operation of replacement rules (see Chapter 5), for graphics functions, and in many other functions and programming constructs. Altogether, more than 120 built-in symbols have `Hold`-like attributes. We now compute lists of the functions having these attributes. (We discuss the construction of the selection inputs in Chapter 5.)

```
Select[Names["System`*"], MemberQ[Attributes[#], HoldAll]&]
```

```
Length[%]
```

As we see, among those functions having the `HoldAll` attribute are `Clear` and `Remove`. If they did not have this attribute, they could not recognize which symbol to clear or remove because their arguments would be evaluated prematurely.

```
Select[Names["System`*"], MemberQ[Attributes[#], HoldFirst]&]
```

```
Length[%]
```

```
Select[Names["System`*"], MemberQ[Attributes[#], HoldRest]&]
```

```
Length[%]
```

The family of `Hold`-like functions has one more member not discussed so far: `HoldAllComplete`. This function is primarily used for typesetting and expression formatting (an important difference between `HoldAll` and `HoldAllComplete` we will discuss shortly).

`HoldAllComplete[`*expr*`]`
is an attribute preventing any evaluation of *expr*.

Currently five built-in functions have the `HoldAllComplete` attribute.

```
Select[Names["System`*"], MemberQ[Attributes[#], HoldAllComplete]&]
```

```
Length[%]
```

The effect of the `Hold`-like attributes `HoldAll`, `HoldFirst`, and `HoldRest` can be overridden with `Evaluate`.

`Evaluate[`*expression*`]`
evaluates *expression*, even if it would otherwise not be evaluated, because it is an argument in a function with `Hold`-type attributes.

It is important to note that `Evaluate` operates only on the current argument, and not on the entire expression. Here is an example.

```
SetAttributes[holdingFunction, HoldAll];

{holdingFunction[1 + 1, a + a],
 holdingFunction[Evaluate[1 + 1], a + a],
 holdingFunction[Evaluate[1 + 1], Evaluate[a + a]],
 Evaluate[holdingFunction[1 + 1, a + a]]}
```

When a function has the attribute `HoldAllComplete`, wrapping `Evaluate` around the arguments will have no effect.

```
SetAttributes[strongHoldingFunction, HoldAllComplete];

{strongHoldingFunction[1 + 1, a + a],
 strongHoldingFunction[Evaluate[1 + 1], a + a],
 strongHoldingFunction[Evaluate[1 + 1], Evaluate[a + a]],
 Evaluate[strongHoldingFunction[1 + 1, a + a]]}
```

The effect of `Hold` can be overridden with `ReleaseHold`.

`ReleaseHold[`*expression*`]`
evaluates *expression*, even if *expression* has the head `Hold` or `HoldForm`.

In the following example, the `Hold` is not overridden because the head of `holdingFunction` is not `Hold` or `HoldForm`, but rather the function `holdingFunction` carries the attribute `HoldAll`.

```
{ReleaseHold[holdingFunction[1 + 1, a + a]],
 holdingFunction[ReleaseHold[1 + 1], a + a],
 holdingFunction[ReleaseHold[1 + 1], ReleaseHold[a + a]]}
```

Here is the right way to use it.

```
ReleaseHold[Hold[1 + 1]]
```

`ReleaseHold` does not work on a `Hold` that lies “deeper” in the expression.

```
ReleaseHold[holdingFunction[Hold[1 + 1]]]
```

It will work only at the part of the expression that `ReleaseHold` encloses.

```
notAHoldingFunction[Hold[1 + 1], ReleaseHold[Hold[a + a]]]
```

It will work only in case the `ReleaseHold` will be evaluated.

```
holdingFunction[Hold[1 + 1], ReleaseHold[Hold[a + a]]]
```

`ReleaseHold` applies at the top level of an expression with nested `Hold`-objects.

```
ReleaseHold[notAHoldingFunction[Hold[1 + 1 + Hold[a + a]]]]

ReleaseHold[notAHoldingFunction[Hold[1 + 1], Hold[a + a]]]
```

`ReleaseHold` does not act nontrivially on functions with the `HoldAllComplete` attribute.

```
SetAttributes[HAC, HoldAllComplete];

HAC[1 + 1] // ReleaseHold
```

Now that we have discussed the attributes of functions, we can explain the differences in the way `Set` and `SetDe`·`layed` work.

```
Attributes[Set]

Attributes[SetDelayed]
```

Both functions leave the left sides (the first argument) unevaluated initially because of their `HoldFirst` and `HoldAll` attribute, which means that a symbol can be found to assign the definition. As this symbol might already have a value, its evaluation must be avoided. However, in contrast to `SetDelayed`, `Set` also computes the right side (the second argument).

Σ (* session summary *) **`TMGBs`PrintSessionSummary[]`**

3.4 Downvalues and Upvalues

For a function definition of the form *function*`[`*insideFunction*`[`*x*`_], ` *y*`_, ` *z*`_] = ` *something*, *Mathematica* associates the definition with the symbol *function*. The definition is a "downvalue" of the function. Especially with basic functions like `Plus` and `Times`, which get used very often, we should not unprotect and add new rules to them. Frequently, in order to save program execution time, we want to associate this definition with the function (symbol) *insideFunction*. This can be done with "upvalues".

`UpSet[`*f*`[`*if*`[`*x*`]], ` *result*`]`
or
f`[`*if*`[`*x*`]] ^= ` *result*
immediately evaluates *result*, and associates it with *if* as a definition.

`UpSetDelayed[`*f*`[`*if*`[`*x*`]], ` *result*`]`
or
f`[`*if*`[`*x*`]] ^:= ` *result*
associates *result* with *if* as a definition, but evaluates *result* only at the time *f*`[`*if*`[`*x*`]]` is called.

Here is an example. `??D` gives all properties of `D`.

```
??D
```

Suppose that, in the future, we want to work with a function `Φ` that has to be differentiated frequently, and even though we know its derivatives, *Mathematica* does not. However, because many other functions also have to be frequently differentiated, we associate our derivative rule with `func`, and not with `D`, to prevent the rule from being tried unnecessarily every time `D` is used.

```
D[Φ[x_], x_] ^:= derivativeOfΦ[x]
```

`D` knows nothing about this new rule.

```
??D
```

`Φ` does know about the rule, however.

```
??Φ
```

It will be applied every time `Φ` is called.

```
D[Φ[newArgument], newArgument]
```

Although the definition is associated with `Φ`, the whole expression including its arguments has to fit.

```
DD[Φ[newArgument], newArgument]
```

A similar approach could be used with the integration of a function that is "transparent" to integration—that is, where the integration can be moved inside to its argument.

$$\int^x \mathtt{transparentFunction}(f(x))\,dx = \mathtt{transparentFunction}\left(\int^x f(x)\right)dx$$

```
Integrate[transparentFunction[f_], x_] ^:=
                  transparentFunction[Integrate[f, x]]

Integrate[transparentFunction[Sin[x]], x]
```

In a somewhat more striking example, the function `headAndArgument` is supposed to give the enclosing head and the enclosed argument. Note the blank after `head` on the left side.

```
head_[headAndArgument[argument_]] ^:= {head, argument}
```

Here is how it works.

```
testHead[headAndArgument[TestArgument]]
```

If an expression has several arguments at the first level, by using `UpSet` and `UpSetDelayed` in function definitions, *Mathematica* associates the corresponding information with each of these arguments. This correspondence (upvalues) works only for arguments at the first level.

`UpSet[`f`[`if_1`[`x`], `if_2`[`x`], …, `if_n`[`x`]], `*result*`]`
or
f`[`if_1`[`x`], `if_2`[`x`], …, `if_n`[`x`]]^= `*result*

immediately evaluates *result* and associates it as a definition with if_1, if_2, …, and if_n.

`UpSetDelayed[`f`[`if_1`[`x`], `if_2`[`x`], …, `if_n`[`x`]], `*result*`]`
or
f`[`if_1`[`x`], `if_2`[`x`], …, `if_n`[`x`]]^:= `*result*

associates *result* as a definition with if_1, if_2, …, and if_n, but *result* is not evaluated until f is called.

For functions with several arguments, the information can be associated with a certain prescribed argument rather than with all arguments at the first level. This association is done with `TagSet` and `TagSetDelayed`.

`TagSet[`*associateWith*`, `f`[`f_1`[`x`], `f_2`[`x`], …, `f_n`[`x`]], `*result*`]`
or
associateWith `/:`f `[`f_1`[`x`], `f_2`[`x`], …, `f_n`[`x`]] = `*result*

immediately evaluates *result* and associates it as a definition with *associateWith* $\in \{f, f_1, f_2, \ldots, f_n\}$.

`TagSetDelayed[`*associateWith*`, `f`[`f_1`[`x`], `f_2`[`x`], …, `f_n` [`x`]], `*result*`]`
or
associateWith `/:`f `[`f_1`[`x`], `f_2`[`x`], …, `f_n`[`x`]] := `*result*

evaluates *result* later and associates it as a definition with *associateWith* $\in \{f, f_1, f_2, \ldots, f_n\}$.

Here is an example in which the following rule is explicitly associated with `x`.

```
Remove[x, y, f]
```

```
x /: f[x, y_] = y;
```

```
??f
```

```
??x
```

If an expression has the form `f[x,` *something*`]`, the rule above is applied.

```
f[x, 3]
```

If `x` is not explicitly the first argument, nothing happens.

```
f[3, x]
```

Here is an example showing the importance of the first level.

```
outside /: outside[middle[inside[xFix]]] = xFixOutside
```

```
middle /: outside[middle[inside[xFix]]] = xFixMiddle
```

An attempt to associate the right-hand side with a symbol at level 2 (or deeper) will fail.

```
inside /: outside[middle[inside[xFix]]] = xFixInside
```

```
xFix /:
outside[middle[inside[xFix]]] = xFixInside
```

Using `TagSet`, we can extend the definition above for the derivative of a function to higher derivatives. (`UpSet` would not have worked because to associate the rule with `func` and the protected symbol `List` from the second argument of `D` would have failed in this case.)

```
Clear[func];
```

```
func /: D[func[x_], {x_, n_}] := derivOfFunc[x, n]
```

`D` has no new rules, but `func` has.

```
??D
```

```
??func
```

Here the new definition is used.

```
D[func[ye], {ye, 23}]
```

If both an upvalue and a downvalue are defined for a given symbol, the definition associated with the upvalue is used before the downvalue definition.

```
Clear[a, b, c, d];
a[b] = c;
a[b] ^= d;
```

```
??a
```

```
??b
```

```
a[b]
```

The order of evaluation of an expression in *Mathematica* will be discussed further at the end of Chapter 4.

Once again, we compare the three functions `Set`, `UpSet`, and `TagSet` in a deeply nested example. We do not discuss each of the outputs in detail because it will quickly become clear what is happening.

```
Clear[a, b, c, d, e];
a[b][c][d] = e
```

```
??a
```

```
??b
```

```
??c
```

```
??d
```

```
??e
```

```
Clear[a, b, c, d, e];
a[b][c][d] ^= e
```

```
??a
```

```
??b
```

```
??c
```

```
??d
```

```
??e
```

```
Clear[a, b, c, d, e];
b /: a[b][c][d] = e
```

```
Clear[a, b, c, d, e];
c /: a[b][c][d] = e
```

In addition to `??`, `DownValues` and `UpValues` can also be used to find out the rules associated with a symbol.

`DownValues[`*function*`]`
gives a list of the downvalues associated with *function*.

`UpValues[`*function*`]`
gives a list of the upvalues associated with *function*.

We now look at what we get for the symbols `outside`, `middle`, and `inside` defined above. The output will involve `HoldPattern` and `:>`, which we shall discuss later, in Chapter 5. Roughly speaking, `HoldPattern` prevents the evaluation of its argument, but at the same time allows pattern matching with this argument and `:>` represents a substitution.

```
Attributes[HoldPattern]
```

Here are the current definition of `outside`, `middle`, and `inside`.

```
DownValues[outside]
```

```
UpValues[outside]
```

```
DownValues[middle]
```

```
UpValues[middle]
```

```
DownValues[inside]
```

```
UpValues[inside]
```

Function definitions can also be input directly in the form `DownValues[…] = …`. We will make use of this possibility from time to time, especially when the order of the definitions is nonstandard.

```
Remove[v];
DownValues[v] = {HoldPattern[v[x_]] :> x};
??v
```

Here is a definition for the symbol $\mathcal{U}$.

```
DownValues[𝒰] =
 {(* specific values *)
  HoldPattern[𝒰[1]] :> 2, HoldPattern[𝒰[-1]] :> -1,
  (* generic case *)
  HoldPattern[𝒰[x_]] :> x^2}
```

The function `DownValues` has the option `Sort`. With the option setting `True`, the special cases are sorted canonically.

```
DownValues[𝒰, Sort -> True]
```

With the option setting `False`, the special cases are returned as they were originally input.

```
DownValues[𝒰, Sort -> False]
```

While for inspecting definitions it is nice to have the various cases sorted, the sorting does take some time. In case one has hundreds of thousands or even million of definitions, for programmatic work on down values, the sorting is frequently not needed and can be avoided to obtain faster algorithms.

At this point in our discussion about `DownValues`, let us make a slightly advanced remark. It will be very useful for later programming applications. The most obvious and important use of `Set` and `SetDelayed` is to make variable assignments and function definitions. In many instances, just a few dozens of them will exist. But it is also possible to have thousands or tens of thousands or even more definitions. They are often for nonpattern ones. The important point is that the time of adding such a definition (with `Set` or `SetDelayed`), the time of removing it (with `Unset`), or the time of its application is basically independent of the number of already existing definitions. The next inputs compare the timings for 100 definitions of `𝕗1` and 1000000 definitions for `𝕗2`. In both cases, the timings are smaller than is the granularity of the `Timing` function.

```
(* create 100 definitions for 𝕗1 *)
Do[𝕗1[i] = i^2, {i, 10^2}]

{(* add a new definition *)
 Timing[𝕗1[101] = 101^2],
 (* remove an existing definition *)
 Timing[𝕗1[100] =.],
 (* apply a definition *)
 Timing[𝕗1[99]]}

(* create 100000 definitions for 𝕗1 *)
Do[𝕗2[i] = i^2, {i, 10^5}]

{Timing[𝕗2[100001] = 100001^2],
 Timing[𝕗2[100000] =.],
 Timing[𝕗2[99999]]}
```

Internally, the definitions are stored in such a way that they can be quickly manipulated and applied. Getting a list of the definitions *itself* via `DownValues` is an operation whose time increases slightly more than linearly with the number of rules. To get a reliable timing for the construction of the list of downvalues of `𝕗1`, we repeat this construction 100 times.

```
Timing[Do[DownValues[𝕗1], {100}]]

Timing[DownValues[𝕗2];]
```

Let us give the following program as a little application of being able to change definitions in constant time (meaning independent of the number of definitions). `randomCrossArray[n]` places "crosses" randomly on a square lattice of size $n \times n$. Each cross occupies five lattice points. The program is carrying out the following process. First, we create a randomly ordered list of all n^2 lattice points. Then, we flag all n^2 crosses as unused by evaluating `Do[unusedSquare⋅Q[squares[[i]]] = True, {i, n^2}]`. This step generates n^2 definitions for the symbol `unusedSquareQ`. Then, we try to put a cross on each of the reordered lattice points. If possible (this means all five lattice points needed for the cross are inside the original $n \times n$ square and none of the five lattice points is not already used for other crosses), we put the new cross in a cross-collecting bag. The five crosses of the new cross are then flagged as used by the line `(unusedSquareQ[#] = False)& /@ newCross`. (Chapter 6 will explain many of the input forms and list-manipulating commands used in this program in more detail.) Finally, `crossGraphics` displays the crosses, each randomly colored.

The next functions `randomPermutation`, `makeCross`, and `randomlyColoredCross` are auxiliary functions needed below.

```
(* load the function RandomPermutation from the package
  "DiscreteMath`Combinatorica`" *)
Needs["DiscreteMath`Combinatorica`"]

(* make a cross around the lattice point {i, j} *)
makeCross[{i_, j_}, n_] :=
Module[{preStar = {{i, j}, {i + 1, j},
                   {i - 1, j}, {i, j + 1}, {i, j - 1}}},
       (* inside the square lattice? *)
       preStar = Select[preStar, Min[#] >= 1 && Max[#] <= n&];
       If[Length[preStar] === 5, cross @@ preStar, $Failed]]

(* make graphics primitive for a colored cross *)
randomlyColoredCross[cross[l_, ___]] :=
{Hue[Random[]], Polygon[l + #& /@
 ({{1, 1}, {3, 1}, {3, -1}, {1, -1}, {1, -3}, {-1, -3}, {-1, -1},
   {-3, -1}, {-3, 1}, {-1, 1}, {-1, 3}, {1, 3}}/2.2)]}

(* display a set of crosses *)
CrossGraphics[crossBag_, n_] :=
Show[Graphics[randomlyColoredCross /@ crossBag],
     PlotRange -> {{1/2, n + 1/2}, {1/2, n + 1/2}},
     Frame -> True, FrameTicks -> False, AspectRatio -> Automatic]
```

The function `randomCrossArray` does the main work and generates the list of randomly placed crosses.

```
randomCrossArray[n_] :=
Module[{squares = Flatten[Table[{i, j}, {i, n}, {j, n}], 1],
        randomlyOrderedSquares, unusedSquareQ, crossBag, newCross},
(* reorder squares *)
randomlyOrderedSquares = squares[[RandomPermutation[n^2]]];
(* mark all squares as unused *)
Do[unusedSquareQ[squares[[i]]] = True, {i, n^2}];
(* a bag to collect crosses *)
crossBag = {};
Do[(* try to put a cross at {i, j} *)
   newCross = makeCross[randomlyOrderedSquares[[i]], n];
   If[(* did the cross still fit? *)
      Head[newCross] === cross,
      If[And @@ (unusedSquareQ /@ newCross),
         (* add cross to cross bag *)
         crossBag = {crossBag, newCross};
         (* mark used squares as used *)
         (unusedSquareQ[#] = False)& /@ newCross]], {i, n^2}];
(* return list of crosses *)
Flatten[crossBag]]
```

The time for the generation of a random cross array is linear in the number of lattice points. The following set of $n = 10, 50, 100, 200$ shows this fact clearly.

```
CrossGraphics[randomCrossArray[ 10],  10] // Timing

CrossGraphics[randomCrossArray[ 50],  50] // Timing

CrossGraphics[randomCrossArray[100], 100] // Timing

CrossGraphics[randomCrossArray[200], 200] // Timing
```

By using packed arrays (to be discussed in Chapter 1 of the Numerics volume [63✱]), the absolute timings for the generation of such random arrays of crosses can be improved, but the complexity (~number of crosses) cannot. (For the average density of the crosses, see [20✱].)

Upvalues can be used to define rules for any head and fixed arguments. Here is an example.

```
𝒜ℬ𝒞 /: _[𝒜ℬ𝒞] := "The upvalue did fire."
```

The definition goes into effect for the head *f*.

```
Clear[f];
f[𝒜ℬ𝒞]
```

The definition goes also into effect for the head `Hold`.

```
Hold[𝒜ℬ𝒞]
```

But in the following input, the `HoldAllComplete` attribute of `HoldComplete` makes sure that the definition for 𝒜ℬ𝒞 does not fire.

```
HoldComplete[𝒜ℬ𝒞]
```

In connection with `UpValues` and `DownValues`, the functions `OwnValues` and `NValues` are also of interest.

`OwnValues[`*symbol*`]`

gives a list of the "direct" values of *symbol*.

`NValues[`*nFunction*`]`
gives a list of all numerical values associated with *nFunction*.

The value assignment to a variable itself can be obtained with `OwnValues`.

```
Remove[x];
x = 4;
{DownValues[x], UpValues[x], OwnValues[x], NValues[x]}
```

To get something as the result of `NValues[`*argument*`]`, the function definition must have either the form `N[`*f*`[`*args*`_]] :=` *numericalValue* or the form `N[`*f*`[`*args*`_],` *digits*`] =` *numericalValue*.

```
Remove[f];
N[f[x_]] := 6.0;
{DownValues[f], UpValues[f], OwnValues[f], NValues[f]}
```

For this special construction, everything is associated with `f` and nothing is associated with `N`.

```
??N
```

```
??f
```

Such a definition works by finding its sole application when a numerical value (`f` in our example) is to be computed *using* `N`.

```
{f[5], f[4] // N, N[f[5]], N @ f[6], f[7.0], N[f[4], 50], f[N[4, 50]]}
```

Here is a definition for *g* that hopefully only works if `N` explicitly gets a second argument.

```
N[g[x_], digits_] := G[N[x, digits]]
```

Again, this definition is stored as an numerical value through `NValues`.

```
??g
```

```
NValues[g]
```

Here is the definition applied.

```
N[g[1/6], 100]
```

Here it is (unexpectedly) too applied.

```
N[g[1/6]]
```

By adding a `Print`-statement to the right-hand side, we see that internally the one-argument call to N expands into a two-argument call with the second argument being the symbol `MachinePrecision`.

```
Clear[g];
```

```
N[g[x_], digits_] := (Print[digits]; G[N[x, digits]])
```

```
N[g[1/6]]
```

Restricting the definition of *g* to calls to `N` with the second argument being integer avoids that N[*g*[1/6]] evaluates nontrivially.

```
Clear[g];
```

```
N[g[x_], digits_Integer] := (Print[digits]; G[N[x, digits]])
```

```
N[g[1/6]]
```

Note that in defining a function with `Set` or `SetDelayed`, the difference between these two should be kept in mind. When using `Set`, the right-hand side is evaluated at the time the definition is made. When using `SetDelayed`, the right-hand side is evaluated when the definition is used. From the standpoint of evaluation of functions, *Mathematica* does not distinguish between the two. Here are two different function definitions.

```
Clear[x, "rc*"];
rc1[x_]  = x^3 + Log[x];
rc2[x_] := x^3 + Log[x];
```

Using `??`, we find out the following about their definitions.

```
?rc1
```

```
??rc2
```

The internal rules used by *Mathematica* to compute `rc1` and `rc2` have the same structure: `HoldPattern[`*leftside*`]` `:>` *rightSide* (as already mentioned, `HoldPattern` and `:>` will be discussed in detail later).

```
DownValues[rc1]
```

```
DownValues[rc2]
```

Using `UpValues` and `DownValues`, we can directly intervene in the internal ordering and form of the storage of function definitions. One use of these functions is for the ordering of definitions. In the Chapters 1 and 2 of the Graphics volume [62✱], we will use the function `DownValues` to directly add and delete rules.

If a function (a symbol) is given as a standalone (that means without arguments), only its `OwnValues` are checked for definitions, not its `DownValues`. Here this is demonstrated.

```
Clear[f, h]
DownValues[f] = {HoldPattern[f] :> h}
```

```
f
```

```
{f[], f[1], f[f]}
```

Now `f` transforms into `h`, because we give a definition to the `OwnValues`.

```
Clear[f];
```

```
OwnValues[f] = {HoldPattern[f] :> h}
```

```
f
```

```
{f[], f[1], f[f]}
```

`SubValues` is one further class of `*Values`.

`SubValues[`*function*`]`
gives a list of the subvalues associated with *function*.

`SubValues` of *function* are values given *function* appears in outermost position within a compound head. Here is an example.

```
Clear[n, g, d]
n[g][d] = n g d
```

```
??n
```

```
{OwnValues[n], UpValues[n], SubValues[n]}
```

```
(* no upvalues or downvalues for g *)
{OwnValues[g], UpValues[g], SubValues[g]}
```

```
(* no upvalues or downvalues for d *)
{OwnValues[d], UpValues[d], SubValues[d]}
```

A further class of definitions can be made concerning the formatting of functions. These definitions are stored in the `FormatValues`. Because we will not discuss formatting, we will not go into detail here.

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

3.5 Functions that Remember Their Values

If a function is recursively defined, it often pays to save values that have already been computed. This process is called dynamic programming or caching.

f[*x*_] := *f*[*x*] = *result*

saves computed values of *f*. This may involve a lot of values, for example, when the function is defined recursively.

We can see how this works by looking at the `FullForm`.

```
FullForm[Unevaluated[f[x_] := f[x] = something]]
```

Note that the use of `Unevaluated` (or any other function with a `Hold`-like attribute)) is needed to really see the `FullForm` of this construction (another function with a `Hold`-like attribute can be used instead of `Unevaluated`). Without it, the definition of `f` would take place immediately because of `SetDelayed`. Just `FullForm` however, does not reveal the process of the definition by itself, because it only displays the result of evaluating its argument.

```
FullForm[f[x_] := f[x] = something]
```

This result means that the implicit grouping used is `f[x_] := (f[x] = something)`. When `f` is called with a concrete value *xConcrete*, after checking the `OwnValues` of `f` to see if `f` evaluates to something, the upvalues of `f` are checked to see if they are applicable. If an upvalue is available, it is used. If no suitable upvalue is available, the downvalue (i.e., the above definition for `f[x_]`) goes into effect. The result of using this function definition is the assignment of a new downvalue to `f`, namely `f[`*xConcrete*`]`. The next time `f[`*xConcrete*`]` is evaluated the stored value of `f[`*xConcrete*`]` will be used and no reevaluation of the general definition of `f` (as defined in `f[x_]:=`...) will be carried out.

We now simulate the recursive integration of the tangent without using the built-in `Integrate` function. For any integer $n \geq 1$,

$$\int^x \tan^n(a\,x)\,dx = \frac{\tan^{n-1}(a\,x)}{(n-1)\,a} - \int^x \tan^{n-2}(a\,x)\,dx$$

$$\int^x \tan(a\,x)\,dx = -\frac{\ln(\cos(a\,x))}{a}$$

For comparison, we now define two functions `TanPowerIntegrate` and `FastTanPowerIntegrate` that carry out the integration above. The second one remembers its values, but the first one does not. Here is our implementation of `TanPowerIntegrate`, which does not remember its values. Note the three pattern variables `a_` (the prefactor), `x_` (the integration variable), and `n_` (the power) and the necessity to define two initial values `n = 0` and `n = 1`.

```
TanPowerIntegrate[Tan[a_ x_]^n_, x_] := (* recursive call *)
                         1/(a (n - 1)) Tan[a x]^(n - 1) -
                         TanPowerIntegrate[Tan[a x]^(n - 2), x];

TanPowerIntegrate[Tan[a_ x_], x_] = -1/a Log[Cos[a x]];

TanPowerIntegrate[1, x_] = x;
```

Here is the result for the antiderivative of $\tan^2(b\,y)$, where we set a = `b`, n = `2`, and x = `y`.

```
TanPowerIntegrate[Tan[b y]^2, y]
```

Differentiation of the last result does not give the original integrand immediately.

```
D[%, y]
```

For comparison, here is the result using the built-in function `Integrate`.

```
Integrate[Tan[b y]^2, y]
```

Differentiating this result also does not appear to produce the original integrand.

```
D[%, y]
```

However, we can apply `Simplify` to obtain the original integrand.

`Simplify[`*expression*`]`

attempts to simplify *expression* by factoring and/or multiplying out. The criterion for simplifying an expression is to minimize `LeafCount[`*expression*`]`.

```
Simplify[%]
```

Note that the `LeafCount` was reduced.

```
{LeafCount[%], LeafCount[%%]}
```

Here are the components forming the parts counted by `LeafCount`, which can be found using `Level[..., {1, Infinity}]`, of the following two expressions.

```
Level[Tan[b y]^2, {0, Infinity}]

Level[-1 + Sec[b y]^2, {0, Infinity}]
```

Caution should be exercised when using `Simplify`. For larger expressions, it can take a great deal of time. A careful application of replacement rules (see Chapter 5) “by hand” (i.e., by application of more specialized *Mathematica* functions like `Expand`, `Factor`, `TrigEx`⸱ `pand`, `TrigFactor`) is often more effective.

Starting from now, we will use `Simplify` from time to time until we discuss the more detailed functions in Chapter 1 of the Symbolics volume [64✱].

We can measure the time a computation takes with `Timing`.

`Timing[`*expression*`]`

gives a list of the CPU time needed for the computation of *expression*, along with the result.

Here is a value of 2^9 and the time needed to compute it. (We see that a limit to the accuracy of the timing measurement exists.)

```
Timing[2^9]
```

In order to avoid looking at the following huge number, we use *expression*`[[1]]` to get just the time.

```
Timing[199999^199999][[1]]
```

Another possibility is to see the huge number, having more than one million digits, and the time needed for its calculation. (The application of the function `N` happens after the timing, so it is not included.)

```
Timing[199999^199999] // N
```

Or we can use a semicolon to suppress the result of the calculation—then only the time is given (and the result `Null`).

```
Timing[199999^199999;]
```

We return now to our integration routine and use `Timing` to measure the time various computations need.

```
Timing[(tanInt500 = TanPowerIntegrate[Tan[c z]^500, z])][[1]]
```

This time is reasonable considering the size of this expression.

```
LeafCount[tanInt500]
```

Here is a part of the whole expression.

```
Short[tanInt500, 5]
```

The computation becomes much faster if we store the results of earlier computations. We achieve this speed-up with the above-described `SetDelayed[Set[..., ...]]` construction.

```
FastTanPowerIntegrate[Tan[a_ x_]^n_, x_] := (* remember *)
FastTanPowerIntegrate[Tan[a x]^n, x] = (* recursive call *)
               1/(a (n - 1)) Tan[a x]^(n - 1) -
               FastTanPowerIntegrate[Tan[a x]^(n - 2), x];

FastTanPowerIntegrate[Tan[a_ x_], x_] = -1/a Log[Cos[a x]];

FastTanPowerIntegrate[1, x_] = x;
```

The first computation takes just slightly more time (because of the additional `Set` statement and the storing of all calculated values) because it has to do the same work. Further integrations become much faster.

```
Timing[FastTanPowerIntegrate[Tan[c z]^500, z]][[1]]
```

For comparison, here are the results for `TanPowerIntegrate`.

```
Timing[TanPowerIntegrate[Tan[c z]^501, z]][[1]]

Timing[TanPowerIntegrate[Tan[c z]^502, z]][[1]]

Timing[TanPowerIntegrate[Tan[c z]^503, z]][[1]]
```

Because the recurrence formula always makes use of the expressions in the previous two steps, the first of the following integrations still takes a relatively long time, because until now only `FastTanPowerIntegrate[Tan[c z]^`*i* with $i = 1 \ldots 498$ and $i = 500$ is already stored. The value for $i = 499$ (which is needed in the computation of `FastTan⋅PowerIntegrate[Tan[c z]^501, z]` still has to be calculated.

```
Timing[FastTanPowerIntegrate[Tan[c z]^501, z]][[1]]
```

Now that all values up to 501 are known, `FastTanPowerIntegrate[Tan[c z]^(`*n*`+1), z]` will compute quickly.

```
Timing[FastTanPowerIntegrate[Tan[c z]^502, z]][[1]]
```

```
Timing[FastTanPowerIntegrate[Tan[c z]^503, z]][[1]]
```

Here are two more examples in which remembering function values pays off. First, we look at a recursive definition of three functions related to each other by the following definitions.

$$f_n = 1^1 f_{n-1} + 2^1 g_{n-1} + 3^1 h_{n-1}$$
$$g_n = 1^2 f_{n-1} + 2^2 g_{n-1} + 3^2 h_{n-1}$$
$$h_n = 1^3 f_{n-1} + 2^3 g_{n-1} + 3^3 h_{n-1}$$

```
Clear[f, g, h];
(* initial conditions *)
f[0] = 1; g[0] = 1; h[0] = 1;
(* the recursion *)
f[n_] := f[n] = 1^1 f[n - 1] + 2^1 g[n - 1] + 3^1 h[n - 1];
g[n_] := g[n] = 1^2 f[n - 1] + 2^2 g[n - 1] + 3^2 h[n - 1];
h[n_] := h[n] = 1^3 f[n - 1] + 2^3 g[n - 1] + 3^3 h[n - 1];

{f[200], g[200], h[200]} // Timing
```

Without remembering the values for `f`, `g`, and `h` from the interlaced definitions, we would have to wait much longer for the values of `f[200]`, `g[200]`, and `h[200]`.

Now, we look at the double recursive definition of the so-called Takeuchi function. (See [35✱], [49✱], [50✱], and [66✱] for a detailed discussion of this function.)

$$t(x,\, y,\, z) = \begin{cases} y & x \le y \\ t(t(x-1,\, y,\, z),\, t(y-1,\, z,\, x),\, t(z-1,\, x,\, y)) & \text{otherwise} \end{cases}$$

(The meaning of the `If` used in the following definition for `Takeuchi` should be obvious; we discuss `If` further in Chapter 5.)

```
TakeuchiT[x_, y_, z_] := TakeuchiT[x, y, z] =
If[x <= y, y,
   TakeuchiT[TakeuchiT[x - 1, y, z], TakeuchiT[y - 1, z, x],
             TakeuchiT[z - 1, x, y]]]

TakeuchiT[14, 13, 0];
```

A whole set of values has been computed.

```
Length[DownValues[TakeuchiT]] - 1
```

The `-1` in the last input accounts for the general definition itself. Here are some of the currently known values of `TakeuchiT`.

```
Short[DownValues[TakeuchiT], 8]
```

Sometimes we want to save only calculated function definitions because their computation may take a lot of time, but not special function values (too many of them may exist). The following construction accomplishes this task. The warning generated by *Mathematica* relates to the atypical appearance of *name*_ on the right-hand side of an assignment. It is a warning message; nothing went really wrong in the following example.

```
Clear[saveSymbolDefinition, n, x];
saveSymbolDefinition[n_, x_Symbol] :=
        saveSymbolDefinition[n, x_] = D[Exp[x^2], {x, n}]
```

Note the `Pattern` construction on the right-hand side of `SetDelayed` and the head specification on the left-hand side. If a function value is to be found immediately, this example fails.

```
saveSymbolDefinition[2, 2]
```

However, we can use an argument with head `Symbol`.

```
saveSymbolDefinition[2, x]
```

Then, a corresponding definition for `saveSymbolDefinition` is available.

```
??saveSymbolDefinition
```

The computation of a special value now proceeds without saving this value. In the next input, the definition for `saveSymbolDefinition[2,x_]` is used.

```
saveSymbolDefinition[2, 2]
```

No rules are stored for `saveSymbolDefinition` with the second argument being numeric.

```
??saveSymbolDefinition
```

The discussed `SetDelayed[Set[... ,...]]` construction by no means requires the two expressions in `SetDelayed` and `Set` to be the same. Here, the last example is implemented with two different symbols.

```
Clear[saveSymbolDefinition, concretSymbolDefinition, x];

saveSymbolDefinition[n_, x_Symbol] :=
         concretSymbolDefinition[n, x_] = D[Exp[x^2], {x, n}]
```

This expression again remains unevaluated because no definition for `concretSymbolDefinition` is available.

```
saveSymbolDefinition[2, 2]
```

A call to `saveSymbolDefinition` with a symbol as the second argument generates a definition for `concretSymbolDefinition`.

```
saveSymbolDefinition[2, z];
concretSymbolDefinition[2, 2];

??concretSymbolDefinition
```

For completeness, let us look at other possible constructions of the form *a* ~ *SetOrSetDelayed* ~ *b* ~ *SetOrSetDelayed* ~ *c*. In addition to the construction `a := b = c`, we also have the two variants `a = b := c` and `a := b := c` (and, of course, the trivial `a = b = c`). They are much less important, however. The first variant leads immediately to a `SetDelayed` assignment. The returned result of the assignment `b:=c` is `Null`, which is the value assigned to `a`.

```
Clear[a, b, c]

a = b := c

??a

??b
```

The second variant `a := b := c` leads to the inside assignment `b := c` only when `a` is called.

```
Clear[a, b, c]

a := b := c

(* make implicit grouping explicit visible *)
Unevaluated[a := b := c] // FullForm

??a

??b
```

```
a
```

```
??b
```

Delayed and immediate assignments can be nested and used for ownvalues, upvalues, downvalues etc. The next input defines a function `f` that assigns an upvalue to the argument of `f`.

```
Clear[f, x, y];
f[x_] := (x /: f[x]  = x^2)
```

Calling now the function `f` with the argument `y` does not change the downvalues of `f`. But it creates an upvalue for `y`.

```
f[y]
```

```
DownValues[f]
```

```
UpValues[y]
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

3.6 Functions in the λ-Calculus

What is a function? According to [10✱], a function is a unique mapping of a set M_1 into a set M_2. Starting with this, A. Church and S. Kleene developed a so-called Lambda Calculus around 1940. One central point in their development was the realization that the name x in a function definition $x \to f(x)$ is arbitrary, which means that we can get rid of it altogether. The function itself is f and $f(x)$ is the value of the function for the argument x. (This was a very rough and simplified version of the whole story. For details of λ-calculus, see [25✱], [61✱], [6✱], [45✱], [51✱], [23✱], [48✱], [65✱], [54✱], [9✱], [37✱], [44✱], and [19✱].) In *Mathematica*, a "pure function" is represented via `Function`.

`Function[`*argument*`,` *map*`(`*argument*`)]`

is the mapping (function) *map* : *argument* → *map*(*argument*). An object with the head `Function` is called a pure function.

Here is an example.

```
f = Function[x, Sin[x]^Exp[x]]
```

We now give the function `f` an argument. We use all three syntactic possibilities to call the function `f` with the argument `1`.

```
{f[1], f @ 1, 1 // f}
```

The variables in the first argument of `Function` are local to `Function`; they have nothing to do with any variables with the same names defined outside. The `HoldAll` attribute of `Function` makes this possible.

```
ξ = 1; Function[ξ, ξ^2]
```

Functions can be nested arbitrarily deep inside one another. Here, we compute sin(π). The argument of the pure function `Function[f, Function[x, f[x]]][Sin]` is `Sin`, and it evaluates to `Function[x, Sin[x]]`. Then, this pure function gets `Pi` as an argument, and the result is 0.

```
Function[f, Function[x, f[x]]][Sin][Pi]
```

What we said earlier about `Set` and `SetDelayed`, concerning assignments to variables used in patterns, also applies to the dummy variables for functions with head `Function`, which means that no assignment is possible to the local variable of a `Function`.

```
Function[x, x = 4; x^2][3]
```

Because the name `x` contains no relevant information, we can drop the name of the function altogether.

`Function[`x`, `$f(x)$`]`
or for several arguments

`Function[{`x_1`, `x_2`, … , `x_n`}, `$f(x_1, x_2, \ldots, x_n)$`]`
or still shorter

f`(#)&`
or for several arguments

f`(#1, #2, … , #`n`)&`

In this example, the argument is `#` and the mapping is arccos(ln(.)).

```
pureFunction = ArcCos[Log[#]]&

pureFunction[1]
```

In the following example, the argument is also a pure function that first replaces the `#` in `f[#[x]]` and then evaluates the resulting `f[#^2&[x]]` to `f[x^2]`.

```
Clear[x, f];

f[#[x]]&[#^2&]
```

Here is an example of a function with two arguments.

```
pureFunctionWith2Arguments = (#1^2 + #2^4)&

pureFunctionWith2Arguments[x, y]
```

Here is the same pure function with two named arguments.

```
Function[{x, y}, x^2 + y^4]
```

Applying it to the arguments `y` and `x` (the order matters) yields $x^2 + y^2$.

```
%[y, x]
```

The pure function `Function[{f, arg}, f[arg]]` applies `f` to `arg`.

```
Function[{f, arg}, f[arg]][Sin, Pi]
```

The next input generates a pure function when applied to the argument `Function`.

```
Function[function, function[#^2]][Function]
```

The pure function of the last output can now be applied to an argument.

```
%[2]
```

Here is the `FullForm` of the function `pureFunction`.

```
FullForm[pureFunction]
```

`#` has been replaced by `Slot`.

`Slot[`*i*`]` or `#`*i*

represents the *i*th formal argument of a pure function. `#0` is the entire pure function.

`SlotSequence[1]` or `##1` or `##`

represents a sequence of all arguments in a pure function definition.

`SlotSequence[`*n*`]` or `##`*n*

represents a sequence of arguments in a pure function definition, starting with the *n*th.

Here is a function that is self-reproducing because of `#0` (in addition it returns its argument).

```
reproduce = {#1, #0}&;
reproduce[1]
```

`##` stands for any possible sequence of arguments. The function `wrapArgumentsInAList` places all of its arguments in a list.

```
wrapArgumentsInAList = {##}&
```

```
wrapArgumentsInAList[1]
```

```
wrapArgumentsInAList[1, 2]
```

```
wrapArgumentsInAList[1, 2, 3]
```

In the following example, the first argument should appear as a squared factor on the right-hand side.

```
useFirstArgumentExtra = (#^2 sinsin[##])&
```

```
useFirstArgumentExtra[fac, rest1, rest2, rest3]
```

Here, we use the remaining arguments, starting with the second argument.

```
useFirstArgumentAndRemainingArgumentsIndividually =
                                    (#^2 sinsin[##2])&
```

```
useFirstArgumentAndRemainingArgumentsIndividually[
                              fac, rest1, rest2, rest3]
```

Here is still another example.

```
(# + #3 + ## + ##2)&[1, 2, 3]
```

The next input explains the result from the last input.

```
1 + 3 + (1 + 2 + 3) + (2 + 3)
```

We can also extract the arguments of the functions.

```
extractArguments = ##&
```

```
extractArguments[1, 2, 3]
```

The last result involved the function `Sequence`.

`Sequence[`a_1`,` a_2`, … ,` a_n`]`

represents a sequence of elements (arguments). The head `Sequence` vanishes as soon as `Sequence[`a_1`,` a_2`, … ,` a_n`]` appears as an argument in a function unless the function has the `HoldAllComplete` or the `SequenceHold` attribute.

Here is such a sequence of arguments.

```
aSequence = Sequence[aa, bb, cc, dd, ee, ff]
```

`List[Sequence[aa, bb, cc, dd, ee, ff]]` causes `Sequence` to disappear.

```
{aSequence}
```

`Sequence` also disappears in nearly any other function (whether specially defined or not).

```
fufu[aSequence]
```

```
Plus[aSequence]
```

If it is nested inside itself, the inside `Sequence` vanishes.

```
Sequence[Sequence[x1, x2], x3, Sequence[x4, x5]]
```

The tendency of `Sequence` to pass on its argument is so dominant that even `Hold`, with its `HoldAll` attribute, has no effect.

```
Hold[Sequence[a, b]]
```

One function strong enough to avoid `Sequence`-objects to disappear is `HoldComplete`.

```
HoldComplete[Sequence[1, 2]]
```

The attribute of `HoldComplete` responsible for this property is `HoldAllComplete`.

```
Attributes[HoldComplete]
```

`Unevaluated` is another function that has the `HoldAllComplete` attribute.

```
Unevaluated[Sequence[1, 2]]
```

`Sequence` also naturally occurs in the following example. We take all arguments, but do not wrap them into an explicitly given function, so that they are returned as a sequence.

```
##&[1, 2, 3]
```

`Sequence` is a very useful function, but it can work in unexpected ways and thus must be used with caution.

If a `Sequence` appears deep inside a held expression, it is not automatically flattened.

```
Remove[𝒢];
SetAttributes[𝒢, HoldFirst];
𝒢[𝒢[Sequence[]], 𝒢[Sequence[]]]
```

Using pure functions, we can generate `f[x]` as follows.

```
Clear[f, x];
```

```
#1[#2]&[f, x]
```

When using pure functions, the ability to provide the argument at various stages in the evaluation of an expression is often possible. All of the following inputs give the same result.

Here, `1 + 1` is evaluated, and then `{g[f[2]]}` evaluates.

```
Clear[g]
```

```
{g[f[#]]}&[1 + 1]
```

Here, `1 + 1` is evaluated, substituted into `g[f[…]]`, and then the outer `List` evaluates.

```
{g[f[#]]&[1 + 1]}
```

Here, `1 + 1` is evaluated, substituted into `f[…]`, and then the outer `{g[…]}` evaluates.

```
{g[f[#]&[1 + 1]]}
```

Here, `1 + 1` is evaluated, and then the outer `{g[f[…]]}` evaluates.

```
{g[f[#&[1 + 1]]]}
```

If the functions `f` and `g` would have definitions, the last four results could be different. Here is an example.

```
Remove[f, g];
SetAttributes[{f, g}, HoldAll]
g[f[2]] = gf;
g[_] = fgl;

{g[f[#]]&[1 + 1]}

{g[f[#&[1 + 1]]]}
```

We turn to the discussion of the attributes of `Function`. `Function` has the attribute `HoldAll`.

```
Attributes[Function]
```

It is necessary for `Function` to have the attribute `HoldAll`. The reason is that the operations carried out in the body of function might not be applicable for a symbol (which is required for the dummy variable of `Function`) or the result might depend on the time in which the calculation is carried out. This `HoldAll` has the following effect: Let `functionsFunction` be a function of one argument that produces a function.

```
functionsFunction[a_] := Function[x, 2 a + x]
```

When given an argument, this output is what we get.

```
functionsFunction[3]
```

Here `2 3` is not evaluated as `6`. (The reason the `x` is renamed `x$` in `Function` will be discussed at the end of the next section in more detail.) Now, if the resulting function is given an argument, everything will be computed.

```
functionsFunction[3][rst]
```

Often, a function will be applied repeatedly, so it would be advantageous not to have to recompute it every time. We can accomplish this state with a function like this.

```
Clear[functionsFunction]

functionsFunction[a_] := Function[x, Evaluate[2a + x]]
functionsFunction[3]
```

We can also accomplish this result with a pure function.

```
{(2 3 #)&, Evaluate[2 3 #]&}
```

But here we must be careful. If the variable used inside `Function` has a value outside it, `Evaluate` does not allow the variable to be screened inside `Function` and is different from the outside, identically named one.

```
ξ = 3;
Function[Evaluate[ξ], ξ^2]
```

Because `#` cannot be named, the form `Function[`*variable*`,` *expression*`]` will generically be needed if several functions are to be nested. When functions with `#` are nested, some attention has to be paid to the brackets. Here is an example in which a function remains in the resulting expression.

```
(# + (#&))&[3]
```

However, the entire expression does not have the head `Function`.

```
%[3]

Head[%%]
```

This result is in contrast to the following example.

```
# + #&[3]
```

The next example is similar. Every `&` denotes a pure function, and so no further argument can be inserted, except by applying the function.

```
fq[#&, #]&[3]
```

To assign an attribute to a pure function (something we are not likely to do often, but sometimes in the following chapters we will make use of `Listable` and `HoldAll` as an attribute of a `Function`), we can use `Function` with a list of attributes.

`Function[{`x_1`, `x_2`,…, `x_n`}, `$f(x_1, x_2, \ldots, x_n)$`, {`*attribute*$_1$`, `*attribute*$_2$`, … , `*attribute*$_m$`}]`
is the pure function $f(x_1, x_2, \ldots, x_n)$ with the arguments $x_1, x_2, \ldots, x_n$ and the attributes *attribute*$_1$, *attribute*$_2$, …, *attribute*$_m$.

In this example, the attribute `Listable` of `Power` is immediately applied by `Power`.

```
Function[p, p^2][{1, 2, 3}]
```

Here, it is not immediately applied.

```
Function[p, newPower[p]][{1, 2, 3}]
```

Here, it is again applied when we add the attribute `Listable` as a third argument to `Function`.

```
Function[p, newPower[p], {Listable}][{1, 2, 3}]
```

Note that the following example does not work.

```
Function[Slot[1], {Listable}][{1, 2, 3}]
```

Only pure functions with named variables allow attributes to be specified.

The application of `Function[`x`, `$f(x)$`]` has a small, usually unimportant side effect: x is added to the list of variables already used.

```
Function[addMeToTheExistingSymbols,
         addMeToTheExistingSymbols^3][3]

??addMe*
```

So although `addMeToTheExistingSymbols` in the last example is a dummy variable, from a programming language point of view the symbol must, of course, be present (in the parsing process).

Using the fact that pure functions can be nested to arbitrary depths, we can efficiently construct very large expressions that consist of several of the same subexpressions without the use of auxiliary variables. For example, suppose we want to calculate the following expression to 100 digits:

$$\frac{(23+31)^3}{34\,564\,534} + \exp\left(\frac{(23+31)^3}{34\,564\,534}\right) + \ln\left(\frac{(23+31)^3}{34\,564\,534} + \exp\left(\frac{(23+31)^3}{34\,564\,534}\right)\right).$$

Here is the direct implementation, followed by a doubly nested pure function to compute this expression.

```
(23 + 31)^3/34564534 + Exp[(23 + 31)^3/34564534] +
     Log[(23 + 31)^3/34564534 +
              Exp[(23 + 31)^3/34564534]] // N[#, 100]&
```

Here is a shorter and more efficient form of the last input.

```
(# + Log[#])&[(# + Exp[#])&[N[(23 + 31)^3/34564534, 100]]]
```

Several repeating variables can be handled by using lists (or other functions that do not compute the arguments) in the intermediate steps. Suppose we want to compute $(3^{33}+2^{22})+(3^{33}+5^{55})+(3^{33}+2^{22})^2+(3^{33}+5^{55})^3$. Here is a direct approach.

```
(3^33 + 2^22) + (3^33 + 5^55) + (3^33 + 2^22)^2 + (3^33 + 5^55)^3
```

Using pure functions, we can use the following input.

```
(#[[1]] + #[[2]] + #[[1]]^2 + #[[2]]^3)&[
          {#1 + #2, #1 + #3}&[3^33, 2^22, 5^55]]
```

However, the following example does not work.

```
(#1 + #2 + #1^2 + #2^3)&[(#1 + #2, #1 + #3)&[3^33, 2^22, 5^55]]
```

Neither does this example, because `Sequence` disappears before the relevant pure function is evaluated.

```
(#1 + #2 + #1^2 + #2^3)&[
    Sequence[#1 + #2, #1 + #3]&[3^33, 2^22, 5^55]]
```

Using `Unevaluated` in this form is also of no help; it has only one argument and the outer function sees only one argument, namely `Unevaluated[…]`, but expects three arguments.

```
(#1 + #2 + #1^2 + #2^3)&[
   Unevaluated[#1 + #2, #1 + #3]&[3^33, 2^22, 5^55]]
```

The following input also does not work. Although the pure function now has the attribute `HoldAllComplete`, `Function` itself does not have this attribute and so removes `Sequence` before going to work.

```
(#1 + #2 + #1^2 + #2^3)&[
    Function[{slot1, slot2, slot3},
       Sequence[slot1 + slot2, slot1 + slot3],
        {SequenceHold}]&[3^33, 2^22, 5^55]]
```

Giving `Function` itself the `HoldAllComplete` attribute makes things work.

```
SetAttributes[Function, HoldAllComplete];
(#1 + #2 + #1^2 + #2^3)&[
    Sequence[#1 + #2, #1 + #3]&[3^33, 2^22, 5^55]]
```

However, it is certainly possible to write the above formulas with the notation `#1, #2` instead of with the more complicated notation `#[[1]], #[[2]]`. The next input uses the construction `ReleaseHold[Hold[Se⋱ quence[…]]]` to generate a sequence of arguments.

```
(#1 + #2 + #1^2 + #2^3)&[
     ReleaseHold[Hold[Sequence[#1 + #2, #1 + #3]]&[
                                       3^33, 2^22, 5^55]]]
```

Another possibility is the use of the command `Apply` (discussed in Chapter 6).

```
Apply[(#1 + #2 + #1^2 + #2^3)&,
      {#1 + #2, #1 + #3}&[3^33, 2^22, 5^55]]
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

3.7 Repeated Application of Functions

Sometimes a function must be applied repeatedly, e.g., in drawing a fractal. The relevant *Mathematica* operations are discussed here.

`Nest[`*function*`,` *start*`,` *numberOfIterations*`]`
applies the function *function numberOfIterations* times to *start*.

Here we apply sin 12 times to $\pi / 7$.

```
Nest[Sin, Pi/7, 12]
```

Note that the following inputs would have given the same result.

```
Nest[Sin[#]&, Pi/7, 12]

Nest[Function[x, Sin[x]], Pi/7, 12]

Nest[Function[Sin[#]], Pi/7, 12]
```

This process goes much faster numerically and yields, of course, a shorter result.

```
Nest[N[Sin[#]]&, N[Pi/7], 45]
```

Here is a somewhat larger example (in terms of the output).

```
Nest[Level[#, {0, Infinity}, Heads -> True]&, Sin[x^2], 2]
```

To collect all intermediate values, `NestList` can be used.

`NestList[`*function*`,` *start*`,` *numberOfIterations*`]`
applies the function *function numberOfIterations* times to *start*, and puts all results in a list, that is, `{`*function*`[`*start*`],` *function*`[`*function*`[`*start*`]],` …`}`.

To illustrate, here are the repeated integrals of the function *f*, starting with $f(x) = 1$.

```
NestList[Integrate[#, x]&, 1, 10]
```

In the following example, the argument (a pure function) is reproduced at every step.

```
NestList[(#&[#])&, #&, 3]
```

To iterate a function with several arguments, we can proceed as follows. It is important that the result has a structure permitting it to serve as an argument of the function. Here, we use a list and extract its element in each iteration step.

```
fz2[x_, y_] := {x^2 + 1, y^2 + 2}

Nest[fz2[#[[1]], #[[2]]]&, {s1, s2}, 6]
```

Using `NestList`, we also get all intermediate results.

```
NestList[fz2[#[[1]], #[[2]]]&, {s1, s2}, 6]
```

Next, we give a little application of `NestList`. Suppose we are given the following iterative mapping [41✱].

$$x_{n+1} = y_n - \operatorname{sign}(x_n)\sqrt{|b\, x_n - c|}$$
$$y_{n+1} = a - x_n$$

Starting at the point {0, 0}, we want to iterate this mapping and look at the first 10000 points $\{x_n, y_n\}$. We will discuss the details of creating plots later.

```
mapPicture[{a_, b_, c_}, {x0_, y0_}] :=
Show[Graphics[{PointSize[0.005], Point /@
NestList[Apply[{#2 - Sign[#1] Sqrt[Abs[b #1 - c]], a - #1}&, #]&,
         {x0, y0}, (* 10000 iterations *) 10000]}],
     PlotRange -> All, AspectRatio -> Automatic]

(* specific values a, b, and c;
  other values give neat pictures too *)
mapPicture[{0.4, 1.0, 1.0}, {0.00, 0.00}]
```

Here is the result for different starting values $\{x_0, y_0\}$.

```
mapPicture[{0.4, 1.0, 1.0}, {0.20, 0.40}]
```

Here is still another plot, this time we also change the values of the parameters a, b, and c and we iterate 20000 times.

```
mapPicture[{1.4, 1.1, 2.2}, {0.20, 0.99}]
```

Here is an animation for $a = \cos(t)$, $b = \sin(t)$, $c = \pi$, and $\{x_0, y_0\} = \{0, 0\}$ as a function of t. The iterated points take on a variety of shapes. We color the points in the order they were generated.

```
coloredMapPicture[{a_, b_, c_}, {x0_, y0_}, o_] :=
Graphics[{PointSize[0.005],
MapIndexed[{(* color in order *) Hue[#2[[1]]/o], Point[#1]}&,
NestList[Apply[{#2 - Sign[#1] Sqrt[Abs[b #1 - c]], a - #1}&, #]&,
         {x0, y0}, o]]}, PlotRange -> All, AspectRatio -> Automatic]

Show[GraphicsArray[(coloredMapPicture[#, {1, 1}, 12000]& /@ #)]]& /@
    Partition[Table[N[{Cos[t], Sin[t], Pi}], {t, 0, 2Pi, 2Pi/15}], 4]
```

The following animation shows the resulting point sets for 230 different values of t.

```
Do[Show[coloredMapPicture[N[{Cos[t], Sin[t], Pi}], {1, 1}, 3000]],
   {t, 0, 2Pi, 2Pi/229}]
```

For a mathematical investigation of such iterated mappings, see [24✱] and [56✱]. (Several other interesting mappings can be found there.)

If the first argument of `NestList` (`Nest`, …) is a pure function and the second argument is a (list of) machine numbers and the third argument is greater than 100, *Mathematica* will often be able to use internal optimizations techniques to carry out the operation in question very quickly. (It will use compiled versions; we will discuss this in detail in Chapter 1 of the Numerics volume [63✱].) Here is an example. Producing a list with 250000 elements of the map

$$x_{n+1} = +y_n + \kappa\cot(x_n)$$
$$y_{n+1} = -x_n - \kappa\tan(y_n)$$

will be carried out in less than one second on a 2 GHz computer.

```
κ = -31/12;
(nl = NestList[{ #[[2]] + κ Cot[#[[1]]], -#[[1]] - κ Tan[#[[2]]]}&,
               N[{51/31, 32/199}],
               250000]); //  Timing
```

Here are points of the list `nl` shown.

```
Show[Graphics[Point /@ nl]]
```

Here is another example of an application of `Nest`. We compute the first few terms in the solution of the ordinary differential equation $y''(x) = -y(x),\ y(0) = 1,\ y'(0) = 0$.

We do this by iteration of the equivalent integral equation $y(x) = 1 - \int_0^x dx' \int_0^{x'} y(x'')\, dx''$ starting with the initial approximation $y_0(x) = 0$ [28✱].

```
rightHandSide[y_] := 1 - Integrate[Integrate[y, {x, 0, ξ}], {ξ, 0, x}]
```

```
NestList[Expand[rightHandSide[#]]&, 0, 7]
```

For comparison, here is the result produced by calculating the first 12 series terms of cos. (The function `Series` will be discussed in Chapter 1 of the Symbolics volume [64✱].)

```
Series[Cos[x], {x, 0, 12}]
```

We could also check the result by substituting it into the original differential equation.

```
D[%%[[-1]], {x, 2}] + %%[[-1]]
```

Here is the function $z \longrightarrow z^z$ ([4✱], [42✱], [26✱], and [70✱]) iterated. (See also Chapter 1 of the Numerics volume [63✱] for a more detailed discussion on this iteration.)

```
NestList[#^#&, N[1 - 2I], 40]
```

Another useful function performing repeated function evaluations is `NestWhileList`.

`NestWhileList[`*function*`,` *start*`,` *test*`,` *compare*`,` *maxIterations*`]`

repeatedly applies the function *function* to *start* until the test *test* no longer gives `True` and returns the list of all calculated elements. The test *test* is applied between the last generated element and the *compare* earlier elements. The function *function* is applied up to a maximum of *maxIterations* times.

If only the last result is of interest, the function `NestWhile` comes in handy.

`NestWhile[`*function*`,` *start*`,` *test*`,` *compare*`,` *maxIterations*`]`

repeatedly applies the function *function* to *start* until the test *test* no longer gives `True` and returns the last calculated element. The test *test* is applied between the last generated element and the *compare* earlier elements. The function *function* is applied up to a maximum of *maxIterations* times.

As an example of the use of `NestWhileList`, let us look at the iterated application of the function $x \longrightarrow 1 + z \ln(x)$, where z is a given parameter. We will iterate until a previously encountered number is encountered again. We limit ourselves to applying the function at most 200 times. (The function `UnsameQ[`arg_1`,` arg_1`,` ..., arg_n`]` gives true only in case all the $\arg_i$ are different. We will discuss this function in Chapter 5.)

```
iteratedList[z_] := NestWhileList[Function[x, N[1 + z Log[x]]],
                                  N[z], UnsameQ, All, 200]
```

Depending on the value of the complex parameter z, the repeated application of $x \longrightarrow 1 + z\ln(x)$ can result in a fixed point.

```
iteratedList[3.]
```

Or it can result in periods of various length. Here is a period of length 3.

```
iteratedList[-2 - 2 I]
```

And here is a period of length 4.

```
Short[iteratedList[6/5 I], 12]
```

The following function calculates the length of the period when given the result from `NestWhileList` as the argument.

```
period[list_] := If[Length[list] === 201, 201,
                    Position[Rest[Reverse[list]],
                             _?(# == Last[list]&), {1}, 1][[1, 1]]]
```

Here we determine the periods 1, 3, and 4 from above.

```
period[iteratedList[3.]]

period[iteratedList[-2 - 2 I]]

period[iteratedList[6/5 I]]
```

In the complex z-plane, the various periods form an interesting pattern. In the following example, we calculate this pattern. To speed up the calculation, we use a compiled version of `Nest`. (We discuss the routine `Compile` in detail in Chapter 1 of the Numerics volume [63✱].)

```
periodCompiled =
Compile[{{z, _Complex}},
        Module[{list, i = 1, last},
               list = NestList[N[1. + z Log[#]]&, z, 100];
               list = Reverse[list];
               last = list[[1]];
               i = 2;
               While[last != list[[i]] && i < 100, i = i + 1];
               i - 1]];

DensityPlot[periodCompiled[x + I y], {x, -3, 3}, {y, -3, 3},
            PlotPoints -> 300, Compiled -> False,
            Mesh -> False, ColorFunction -> Hue]
```

Here is another example of the use of `NestWhileList`. We apply the function `1/# - IntegerPart[1/#]&` to a complex number z until we find an already earlier encountered value. (We visualized the map `1/# - IntegerPart[1/#]&` in Subsection 1.2.2.) We use rational complex numbers as starting values and display the length of the resulting lists (this means the sum of the length of the initial and the periodic part) as a density plot in $[0, n] \times [0, n]$.

```
cF[x_] := NestWhileList[1/# - IntegerPart[1/#]&, x, UnsameQ, All]

n = 500;
Off[Power::infy]; Off[Infinity::indet];
(* square array of data points *)
data = Table[Length[cF[i/n + I j/n]], {i, 0, n}, {j, 0, n}];

ListDensityPlot[data, Mesh -> False, FrameTicks -> None,
                ColorFunction -> (GrayLevel[1 - #]&),
                PlotRange -> All]
```

For situations in which the result approaches an asymptotic value, we can use `FixedPointList`.

`FixedPointList[`*function*`, `*start*`, `*maxIterations*`]`

repeatedly applies the function *function* to *start* until the result stops changing, up to a maximum of *maxIterations* times, and puts all of the results in a list.

The detailed meaning of "the result stops changing" is specified with the option `SameTest`.

```
Options[FixedPointList]
```

This option will be discussed in detail in Chapter 6. Note that only two consecutive values are compared! With the default setting used above, the result "stops changing" when successive results are identical, up to the last digits. We discuss the issue " being identical" in Chapter 1 of the Numerics volume [63✱] in more detail.

`FixedPoint[`*function*`, `*start*`, `*maxIterations*`]`

repeatedly applies the function *function* until the result stops changing, up to a maximum of *maxIterations* times, and outputs the unchanging result (the fixed point) satisfying *function*(*arg*) = *function*(*function*(*arg*)).

We now use Newton's method to find the square root of the number c. This goal involves solving $f(x) = x^2 - c$ iteratively. The general Newton method is based on this iteration:

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}.$$

For finding the square root, the iteration reduces to

$$x_{i+1} = x_i - \frac{x_i^2 - c}{2\,x_i} = \frac{x_i}{2} + \frac{c}{2\,x_i}.$$

Amazing accuracy is obtained after just a few iterations. Here is the square root of $c = 3$.

```
FixedPointList[Function[x, x/2 + 3/(2x)], N[1, 100]]
```

This result is quite precise.

```
3 - %[[-1]]^2
```

For some interesting observations about the last iteration, see [16✱], [36✱]; x_n can be expressed in closed form through the starting value x_0 by $x_n = \sqrt{c}\left(1 + \left(\left(x_0 - \sqrt{c}\right)/\left(x_0 + \sqrt{c}\right)\right)^{2^n}\right) \Big/ \left(1 - \left(\left(x_0 - \sqrt{c}\right)/\left(x_0 + \sqrt{c}\right)\right)^{2^n}\right)$ [69✱], [14✱]. (For optimal starting values of the Newton iterations, see [55✱].)

For higher order polynomials, the Newton iteration exhibits some very interesting features. One of them is the answer to the question: As a function of the starting value, to which root will the solution converge? The following picture shows the convergence to the roots of $z^5 = 1$ as a function of the complex start value. (For details on the basins of attractions of the Newton iteration, see [74✱], [33✱], [67✱], [31✱], [22✱], [11✱], and [43✱]; for a choice of starting values that reach all roots, see [30✱].)

```
newton5[z_] = z - (z^5 - 1)/D[z^5 - 1, z];

data = Table[FixedPoint[newton5, N[x + I y]],
             {y, -2, 2, 4/299}, {x, -2, 2, 4/299}];

ListDensityPlot[Im[data], Mesh -> False, ColorFunction -> Hue,
                MeshRange -> {{-2, 2}, {-2, 2}}]
```

Other interesting fractals can be obtained from Newton iterations. The next graphic shows the number of iterations needed until a fixed point is reached.

```
lfpl[x_] := Length[FixedPointList[Function[z, (1/z^4 + 4z)/5], x]]
```

```
pp = 401;
data = Table[lfpl[x + I y], {y, -1., 1., 2/pp}, {x, -1., 1., 2/pp}];
```

```
ListDensityPlot[data, Mesh -> False, ColorFunction -> (Hue[3 #]&),
                MeshRange -> {{-2, 2}, {-2, 2}}]
```

Here is another little application of FoldList. Given a univariate polynomial p and a complex number z, we form the Cantor series [58✱] defined as (the square brackets in $C[p](z)$ indicate the functional dependence on the polynomial p)

$$C[p](z) = \sum_{n=1}^{\infty}\left(\prod_{k=1}^{\infty} c_k\right)^{-1}$$
$$c_k = p(c_{k-1})$$
$$c_1 = z.$$

Here is a polynomial $p(z)$.

```
p[z_] := -10 + 6 z - 10 z^2 - 10 z^3 - 7 z^4 - 3 z^5 +
          5 z^7 - 8 z^8 - 4 z^9 + 6 z^10 + z^11 - 4 z^12
```

The function step updates the three-element list {*term*, *product*, *sum*}. Here term stands for $p(c_{k-1})$, *product* for $\left(\prod_{k=1}^{\infty} c_k\right)^{-1}$ and sum for $C[p](z)$.

```
step[{term_, product_, sum_}] :=
      {#, product/#, sum + product/#}&[p[term]]
```

As a function of the initial z, the function CantorSeries adds terms as long as they change the cumulative sum. (To terminate the repeated application of step we use a numerical z.)

```
CantorSeries[z_] :=
FixedPoint[step, {z, 1/z, 0}, SameTest -> (#1[[3]] == #2[[3]]&)][[3]]
```

Here is a plot of $C[p](z)$ over the complex z-plane.

```
Plot3D[Re[CantorSeries[N[x + I y, 20]]], {x, -2, 2}, {y, -2, 2},
       PlotPoints -> 400, Mesh -> False, ClipFill -> None,
       PlotRange -> {-0.1, 0.1}]
```

By using FixedPointList instead of FixedPoint we can easily count the number of terms needed in $C[p](z)$.

```
CantorSeriesList[startTerm_] :=
FixedPointList[step, {startTerm, 1/startTerm, 0},
               SameTest -> (#1[[3]] == #2[[3]]&)]
```

The following graphic shows the number of terms as a function of the initial z. This time we use the polynomial $p(x) = x^3 - x^2 + x - 1$.

```
p[x_] := x^3 - x^2 + x - 1;
```

```
Plot3D[Length[CantorSeriesList[N[x + I y, 20]]],
       {x, -3, 3}, {y, -3, 3}, PlotPoints -> 400,
       Mesh -> False, ClipFill -> None, PlotRange -> All]
```

For a function of two arguments, the commands Fold and FoldList are useful for repeatedly applying the function.

`FoldList[`*function*`, ` *x*`, {`a_1`, `a_2`, … , `a_n`}]`
forms the list `{`*x*`, ` *function*`[`*x*`, `a_1`], ` *function*`[`*f*`[`*x*`, `a_1`], `a_2`]…}`. Here, *function* must be a function of two arguments.

`Fold[`*function*`, ` *x*`, {`a_1`, `a_2`, … , `a_n`}]`
gives the last element of `FoldList[`*function*`, ` *x*`, {`a_1`, `a_2`, … , `a_n`}]`.

Suppose we want to raise an expression to a series of different powers. This goal can be accomplished with `FoldList` in pure function form.

```
FoldList[Power[#1, #2]&, β, {exp1, exp2, exp3, exp4, exp5, exp6, exp7}]
```

In the next example, the imaginary unit *i* is successively raised to exponents that are multiples of *i*.

```
FoldList[Power, I, {I, 2 I, 3 I, 4 I, 5 I, 6 I, 7 I}]
```

In the following examples, we clearly see how the use of numeric rather than symbolic variables saves time. In the first example, `N` is applied only after all elements in the list have been symbolically computed.

```
FoldList[Power, I, {I, 2 I, 3 I, 4 I, 5 I, 6 I, 7 I, 8 I,
                    9 I, 10 I, 11 I, 12 I, 13 I}] // N
```

Here is what happens when numerical values are computed inside of `FoldList`. Clearly, it will result in a significant savings in time.

```
FoldList[N[Power[#1, #2]]&, I,
         {I, 2I, 3I, 4I, 5I, 6I, 7I, 8I, 9I, 10I, 11I, 12I, 13I}] // N
```

`FoldList` and `Nest` can be used, along with appropriate pure functions, to construct short and fast expressions that do complex work. For example, we calculate the first *n* partial products of the expansion of $\sqrt{z}$ around $z = 1$ [72✱], [40✱]. The expansion coefficients can be calculated using the following recursion.

$$\sqrt{1+z} = \prod_{k=1}^{\infty} \frac{2\,a_k(z)+2}{a_k(z)+1}$$

$$a_1(z) = z$$

$$a_k(z) = \frac{a_{k-1}^2(z)}{4\,a_{k-1}(z)+4}$$

```
sqrtApproximationList[z_, n_] :=
       Rest[FoldList[Times, 1, (2# + 2)/(# + 2)&[
            NestList[#^2/(4# + 4)&, z - 1, n]]]]
```

Here is one example.

```
sqrtApproximationList[2, 7]
```

The product converges quickly.

```
N[%]
```

Using high-precision arithmetic, we can see it calculate the difference to the exact value (the outer `N` prevents display of all 500 digits; the `Off` used in the next input will be discussed in Chapter 4).

```
Off[N::meprec];
N[N[%%% - Sqrt[2], 500]]
```

`Fold` is a very useful construction that allows for the use of “varying parameters” in the steps of an iterative calculation. In the following example, the third argument of `FoldList` controls the decreasing diameter of the rings. (Ignore the details of the graphics construction for the moment.)

```
(* calculating a new set of orthogonal directions *)
step[{mp_, n_, p_}, r_] :=
Module[{newn = Cross[n, p], b}, (* newn and b are new directions *)
        b = Cross[newn, p]; (* hexagon in plane of new directions *)
        Table[{mp + r #, newn, #}&[Cos[t] p + Sin[t] b],
              {t, 0.0, 2. Pi, 2. Pi/6}]]

Show[Graphics3D[
MapIndexed[{Thickness[0.015/#2[[1]]], Hue[#2[[1]]/7],
            (* color according to size *) Line[First /@ #1]}&,
             (* make many tori of different size using Fold *)
             FoldList[Function[{x, y}, Map[step[#, y]&, x, {-3}]],
                        (* rotation matrices *)
                        Table[{{Cos[t], Sin[t], 0},
                               {0     , 0     , 1},
                               {Cos[t], Sin[t], 0}},
                              {t, 2.Pi/6, 2.Pi, 2.Pi/6}],
      (* three different sizes *) {1/3, 1/6, 1/12}], {-4}]],
      PlotRange -> All, Boxed -> False, ViewPoint -> {1.3, -1.4, 1.8}]
```

If only one argument exists, but many functions (heads) should be applied one after another, we should use `ComposeList`.

> `ComposeList[{`f_1`,` f_2`, …,` f_n`},` *arg*`]`
>
> forms `{`*arg*`,` f_1`[`*arg*`],` f_2`[`f_1`[`*arg*`]], …}`.

`ComposeList` is a generalization of `NestList`. Here is a simple example involving `ComposeList`.

```
ComposeList[(* a list of seven (pure) functions *)
            {Sin, Sin[#]&, Times[#, #]&, Log[5 #]&, 6#&, 5 + #&,
             Function[x, x^2]}, aabbcc]
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

3.8 Functions of Functions

Given functions f_1, f_2, … f_n, we can combine them in many ways (and later apply them to arguments). One possibility is to apply them one after another f_1, f_2, …, f_n using `Composition`.

> `Composition[`f_1`,` f_2`, …,` f_n`]`
>
> leads to the application of the f_i, one after another.

`Composition` can be regarded as `ComposeList` without an argument. It works as follows.

```
co = Composition[# + 1&, # + 2&, # + 3&]

co @ 𝒯
```

Compositions are not immediately carried out.

```
Composition[Sin, ArcSin]
```

Here are `Plus` and `Times` used with one argument. In this case, they evaluate just to their argument.

```
Composition[(#^2 + 2)&, Plus, Sqrt, Times, Sin]
```

When applied to an argument, the composition is actually carried out.

```
%[aaa]
```

We can make a function *funcFunc* acting only on the function *func* in *func*[*arg*] (i.e., on the head *func*) but not on the argument *arg*. This result is accomplished with `Operate`.

> `Operate`[*funcFunc*, *func*[*arg*]]
> gives (*funcFunc*[*func*])[*arg*].

Here is a simple example.

```
Operate[fOuter, fInner[4]]
```

Here is a slightly more complicated example.

```
Remove[a, c, f, x];
Operate[a, (c a)[f[x]]]

FullForm[%]
```

To apply a function of the type just obtained to an argument, we can use the command `Through`.

> `Through`[*funcFunc*[f_1, f_2, …, f_n][*arg*]]
> gives (*funcFunc*[*func*])[*arg*].

If the head of a function is composite, the function is not immediately applied.

```
(Sin + Cos + Tan + Cot)[Pi/4]
```

To make this sum operate on `Pi/4`, we need `Through`.

```
Through[(Sin + Cos + Tan + Cot)[Pi/4]]
```

Here are the single terms of this sum.

```
{Sin[Pi/4], Cos[Pi/4], Tan[Pi/4], Cot[Pi/4]}
```

In this case, we could get the same result with the following, slightly less convenient construction.

```
Unprotect[Plus];
(Sin + Cos + Tan + Cot)[x_] := Sin[x] + Cos[x] + Tan[x] + Cot[x];

Protect[Plus];
(Sin + Cos + Tan + Cot)[Pi/4]
```

Unfortunately, with a minus sign, we get another useless result, which is explained by looking at the `FullForm` of expressions of the form −*expr*.

```
Through[(Sin + Cos + Tan - Cot)[Pi/4]]

FullForm[Sin + Cos + Tan - Cot]
```

After applying `Through` to the expression `Operate[a, (c a)[f[x]]]`, we get the following result.

```
Operate[a, (c a)[f[x]]] // Through
```

The inverse of a given function is an especially interesting new function, which can be obtained with `InverseFunc`⋱
`tion`.

```
InverseFunction[function]
```
finds the inverse function for *function*, so that `InverseFunction[function][x]` $= x$.

Whenever possible, the inverse function is given explicitly.

```
InverseFunction[Sin]
```

```
InverseFunction[Tan[#]&]
```

If this is not possible, the result remains in symbolic form.

```
InverseFunction[fufufu]
```

These are the functions that *Mathematica* can invert. (We discuss the meaning of the individual commands used in the next input in the next chapters.)

```
Drop[DeleteCases[(* select all built-in functions that have
                    a value for its inverse function *)
DeleteCases[First[#]& /@ DownValues[InverseFunction],
            HoldPattern | Literal | InverseFunction,
            {0, Infinity}, Heads -> True], _Integer], {-1}]
```

Pure functions are not “explicitly inverted”.

```
InverseFunction[3 # + 7&]
```

```
InverseFunction[Sin[#]&]
```

```
InverseFunction[Function[x, Sin[x]]]
```

```
InverseFunction[(2 + 1)&]
```

Inverse functions can be differentiated and integrated.

```
D[InverseFunction[f][x], x]
```

```
D[InverseFunction[f][x], {x, 2}]
```

The ($n = 5$)th derivative of $f^{(-1)}(x)$ is proportional to $f'\big(f^{(-1)}(x)\big)^9$ [3✱], [32✱]. This means, that by multiplying with $f'\big(f^{(-1)}(x)\big)^9$ we obtain a polynomial.

```
f'[InverseFunction[f][x]]^9 D[InverseFunction[f][x], {x, 5}] //
                                                          Expand
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

Overview

In this chapter we have discussed attributes and options. Because of space limitations in this overview, we do not give all commands that can carry a given attribute. (We come back to this issue in Chapter 6.)

```
Get[ToFileName[ReplacePart[
          "FileName" /. NotebookInformation[EvaluationNotebook[]],
           "ChapterOverview.m", 2]]];
```

```
ChapterOverview["Programming", 3]
```

Exercises

1.[L1] Predictions

What will be the results of the following *Mathematica* inputs?

a)
```
#^2&[1/#^3&[2]]
Function[#, #^2][x]
Function[Slot, Slot^2][x]
Function[Slot, #^2&[Slot]][5]
```

b)
```
sin[##]&[1, 2]
#[[1]][#[[0]]]&[C[Print]]
fun[SlotSequence[1 + 1]]&[1, 2]
(Slot[Slot[1]])&[1]
```

c)
```
fg[x_Integer] := {x, Integer}
fg[x_Times] := {x, Times}
fg[x_Rational] := {x, Rational}
fg[x_Divide] := {x, Divide}
fg[x_] := {x, arbitrary};
{fg[3],
 fg[Unevaluated[3/1]],
 fg[Unevaluated[3]],
 fg[Divide[3, 1]],
 fg[Unevaluated[Divide[3, 1]]],
 fg[Unevaluated[Rational[3, 1]]]}
```

d)
```
f[x_x] := x[[1]]
f[x[3]]
f[x[x]]
f[x]
Clear[f];
f[Head_] := Head[Head]
f[Sin[Cos]]
```

e)
```
f[Symbol_Symbol] := Symbol
f[Sin]
```

f)
```
f[Integer_] := Integer
f[3]
f[x]
```

g)
```
clock[Print[4]; #]&[Print[3]; #]&[Print[2]; #]&[Print[1]; #]&[hand]
```

h)
```
#1&[]
#2&[0]
#0&[]
##&[]
##0&[]
###&[2, 3, 4]
```

```
  xa&b&c&[d]
  (#&)&[2][3]
  (((#&)&)&)[1][2][3]
```

i) `Evaluate[D[#, x]]&`

j)
```
f[x_Pattern] := x^2
f[x_Integer] = f[x_Integer]
f[3]
```

k)
```
f[x_] := Evaluate[Expand[x]]
f[(x + y)^3]
```

l) `2 // ((1 ~ #1 + #2& ~ #1)& @ #1)&`

m) `a = x_y; f[a] := x^2; f[a]`

n) `N[1[1]] N[1[1]]`

o)
```
f[x_Blank] := x^2;
f[x_y[x]] := x^-2;
f[x_(_y)] := x
f[Pattern[x, Blank[Blank[y]]]] := x^-1
f[_] + f[y[z]] + f[y[z][x]] + f[_head] +
f[Blank[y]] + f[Blank[y][1]] + f[2 y[5]]
```

p) `Flat[Flat[Flat, Flat]]`

q)
```
v := (Remove[a]; 1); a = 2;
v + a
Remove[a, v];
v := (Remove[a]; 1); a = 2;
a + v
```

r)
```
Function[x, x] - Function[y, y]
Function[Function, #^2&][Depth]
```

s)
```
f[x_] := (f[Evaluate[Pattern[y, Blank[Head[x]]]]] := y + Head[x]; f[x^2])
f[2]
```

t)
```
f[x_][y_] := f[y][x]
f[1][2]
```

u)
```
Function[functionBody, Function[s, functionBody][3]][11 s + 111 s^11]

Function[{functionArg, functionBody}, Function[functionArg,
                       functionBody][3]][s, 11 s + 111 s^11]
```

v) `Function[f, (# f)&[3]][#]`

w) `# & & & & & & [1][2][3][4][5][6]`

x)
```
noGo[x_] := (x = 11)
myNewVar = 1; noGo[Unevaluated[myNewVar]]

Remove[noGo]
SetAttributes[noGo, HoldFirst]
noGo[x_] := (x = 11)
myNewVar = 1; noGo[myNewVar]
myNewVar = 1; noGo[Unevaluated[myNewVar]]

myNewVar = 1; noGo[Evaluate[myNewVar]]
```

y)
```
Function[c, Slot[c] SlotSequence[c]&[1, 2, 3], Listable][{1, 2, 3}]

Function[Slot, Slot[1]]&[C][1][1] - Function[Slot, Slot[1]][C]
```

z)
```
Slot[1/2 + 1/2]&[1, 2]

k = 1; k = 2; (#1 - #k) + (#2 - #k)
```

2.^{L2} $(a+b)^{2n+1}$, Laguerre Polynomials

a) Expressions of the form $(a+b)^{2n+1}$ can be written in the following way, at least for the odd powers given here:

$$\begin{aligned}(a+b)^3 &= a^3+b^3+3\,a\,b\,(a+b)\left(a^2+a\,b+b^2\right)^0\\ (a+b)^5 &= a^5+b^5+5\,a\,b\,(a+b)\left(a^2+a\,b+b^2\right)^1\\ (a+b)^7 &= a^7+b^7+7\,a\,b\,(a+b)\left(a^2+a\,b+b^2\right)^2,\end{aligned}$$

Program a function that tries to write expressions of the form $(var_1+var_2)^{exp}$ in this way, and determine whether this can be done for $(a+b)^9$, $(a+b)^{11}$, …, etc.

b) Use the symbolic formula [73✸], [17✸]

$$L_n^{(a)}(z)=\frac{(-1)^n}{n!}\exp\left(-\frac{\partial}{\partial z}\,z\,\frac{\partial}{\partial z}+a\,\frac{\partial}{\partial z}\right)z^n$$

to derive explicit forms of the first few ($n = 0, 1, \ldots, 5$) Laguerre polynomials $L_n^{(a)}(z)$.

3.^{L1} $\frac{d}{d\,a\,x}\int^{a\,x} f(y)\,dy$

Given a function of the form `f[x_] :=` *notAnalyticallyIntegrable*, examine the results of `D[Integrate[f[x], x], x]` and `D[Integrate[f[a x], a x], a x]`. How must `D` and/or `Integrate` be modified to get the desired results in the second case?

4.^{L1} `Pattern[name, _]`

a)
```
Φ[Pattern[2, _]] = 2^2;
??Φ
{Φ[2], Φ[3], Φ[Pattern[2, _]]}
```

b)
```
Φ[Pattern[I, _]] = I^2;
??Φ
{Φ[2], Φ[I]}
```

c)
```
Φ[Pattern[I, _]] := I^2;
??Φ
{Φ[2], Φ[I]}
```

d)
```
Φ[Pattern[a[2], _]] := a[2]^2;
??Φ
{Φ[2], Φ[Pattern[a[2], _]]}
```

d) `FullForm[_[_]]`

5.^{L1} Puzzles

What could the input `In[1]` be to get the following two sets of inputs and outputs?

a)

```
In[2]:= b[c]
Out[2]= b[c]
In[3]:= Head[b[c]]
Out[3]= d
```

b)

```
In[2]:= Remove[f, x]
In[3]:= f[x_] := x^2
In[4]:= f[2]
Out[4]= 16
```

c) Find a *Mathematica* expression *expression* for which *expression*`[[1, 1]]` and *expression*`[[1]][[1]]` are different.

d) Given the definition `f[x_Real] := x^2`, can one give any argument *arg* that is not a real number, such that `f[`*arg*`]` evaluates to its square?

e) What will the result of this input be?

```
FixedPoint[Head, arbitraryExpression]
```

f) What will the results of the following three inputs be?

```
Function[{s}, OIOI[s]][Unevaluated[κ]]

Function[{s}, OIOI[s]][Unevaluated[Unevaluated[κ]]]

Function[{s}, OIOI[s]][Unevaluated[Unevaluated[Unevaluated[κ]]]]
```

g) What will be the result of the following input?

```
DirectedInfinity[Infinity[[1]]]
```

h) What could have been the input `In[1]` to get the following inputs and outputs?

```
In[2]:= a^2
Out[2]= b
In[3]:= Clear[a]
Out[3]= a
In[4]:= Remove[a]
Out[4]= a
```

i) Predict the result of the following input.

```
f[_] := (f[_] := #0[# + 1]; # + 1)&[1]

f[f[f[f[f[2]]]]]
```

6.^{L2} Different Patterns

For the following function definitions, find realizations for which the corresponding pattern matches.

a) `f[x_, a[b_, c_]_] := 𝕗[x, a, b, c]`

b) `f[x_, a_[b_, c_]] := 𝕗[x, a, b, c]`

c) `f[a_ b_ c_] := 𝕗[a, b, c]`

d) `f[_ _] := 𝕗`

7.^{L1} `Plot[`*numberFunction*`]`

Define a function $f(x)$ that gives 3 for integer arguments, 2 for rational arguments, and 1 for real arguments. Try to plot this function using `Plot[f[x], {x, -3, 3}]`. What can one conclude from this attempt to make a `Plot`?

8.^{L1} Tower of Powers

What is the limit value of the following power tower?

$$\left(\sqrt{2}\right)^{\left(\sqrt{2}\right)^{\left(\sqrt{2}\right)^{\left(\sqrt{2}\right)^{\cdots}}}}$$

Calculate numerical values for the first few iterations.

9.^{L1} Cayley Multiplication

Implement the (associative) Cayley multiplication 𝒞𝒯 (short for `Caley𝒯imes`). This operation is binary with the following multiplication table.

	a	***b***	***c***	***e***
a	*e*	*c*	*b*	*a*
b	*c*	*e*	*a*	*b*
c	*b*	*a*	*e*	*c*
e	*a*	*b*	*c*	*e*

Find the following result.

```
𝒞𝒯[a, b, c, a, c, e, a, c, b, b, c, a, e, a, c, c, a, b, a, c,
   a, c, a, e, b, b, a, a, e, c, b, b, a, a, c, e, e, e, a, a,
   b, b, b, a, b, c, b, c, a, a, c, c, c, b, a, a, e, e, c]
```

How often are the various multiplication rules applied?

Solutions

1. Predictions

We evaluate the various inputs and comment on the results.

a) First, the pure function $x \to 1/x^3$ is applied for $x = 2$, and then the pure function $x \to x^2$ is applied. The result is $\left(1/2^3\right)^2 = 1/64$.

```
#^2&[1/#^3&[2]]
```

This input does not work.

```
Function[#, #^2][x]
```

It does not work because the first argument of `Function` must be a symbol or a list of symbols. But `#` is not a symbol.

```
FullForm[#]
```

```
Head[#]
```

Now, it works because `Slot` without any arguments is, of course, a symbol.

```
Function[Slot, Slot^2][x]
```

In the last input, `Function[Slot, #^2&[Slot]][5]`, we get the result `5[1]`^{2}.

```
Function[Slot, #^2&[Slot]][5]
```

Using `FullForm`, we see all occurrences of `Slot`.

```
Hold[Function[Slot, #^2&[Slot]][5]] // FullForm
```

Because `Slot` is the dummy variable of `Function`, its name does not matter.

```
Function[slot, slot[1]^2&[slot]][5]
```

The evaluation proceeds by first substituting the 5 for `Slot`, which yields `5[1]^2&[5]`. The pure function `5[1]^2&` gets applied to the argument 5 and finally yields `5[1]`^{2}.

```
5[1]^2&[5]
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

b) `sin[##]&` is a pure function of one or several arguments. Because no general rules for `sin` exists, if we apply it to the argument, we get `sin[1, 2]`.

```
sin[##]&[1, 2]
```

`#[[1]][#[[0]]]&[C[Print]]` is a slightly more complicated example. First, the `#`s are replaced with `C[Print]`. Then, `Part` comes into effect and we get `Print[C]`, which prints `C` as a result.

```
#[[1]][#[[0]]]&[C[Print]]
```

Because of the attribute `HoldAll` of `Function`, `1 + 1` is not evaluated and `SlotSequence[1 + 1]` is not a valid `SlotSequence`-object. Thus, it all remains unevaluated.

```
fun[SlotSequence[1 + 1]]&[1, 2]
```

In `(Slot[Slot[1]])&[1]`, nearly the same thing happens. `Slot[Slot[1]]` is not an allowed `Slot`-object; its argument must be an integer. The argument of `Slot` must be a nonnegative integer.

```
(Slot[Slot[1]])&[1]
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

c) For the given function definition, it is clear that `fg[3]` is `{3, Integer}`. In the second case, `fg` applies to `Times[3, Power[1, -1]]` (not to `Divide[3, 1]` and not to `3`), and so, we get the result `{3, Times}`. The third case is like the first one. In the fifth case, the `Divide` that appears explicitly in the construction `Unevaluated[Divide[...]]` plays a role for the first time; in the fourth case, we divide first, and thus get 3. In the last case, `Unevaluated` prevents the simplification in `Rational[3, 1]`, and so the result is `{3, Rational}`.

```
fg[x_Integer] := {x, Integer}
fg[x_Times] := {x, Times}
fg[x_Rational] := {x, Rational}
fg[x_Divide] := {x, Divide}
fg[x_] := {x, Egal};
{fg[3],
 fg[Unevaluated[3/1]],
 fg[Unevaluated[3]],
 fg[Divide[3, 1]],
 fg[Unevaluated[Divide[3, 1]]],
 fg[Unevaluated[Rational[3, 1]]]}
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

d) The `x` is regarded as a pattern that stands for an arbitrary argument with head `x`. This is the reason for the extraction of the first part in the first two examples. The `x` appearing on the right-hand side—that is, the `x` to the left of _ is the first argument from `Pattern`. For an argument `x` of `f` with head `Symbol`, we have not defined anything, and `f[x]` remains unevaluated.

```
f[x_x] := x[[1]]
```

```
f[x[3]]
```

```
f[x[x]]
```

```
f[x]
```

In both appearances, `Head` on the right is regarded as a local variable, and not as the command `Head`. The functional meaning of `Head` would apply at a time when the right-hand side of the definition would be evaluated. At this time, `Head` is already replaced by the actual realization of the pattern called `Head`, which is the reason for the result *arg*`[`*arg*`]`.

```
Clear[f];
f[Head_] := Head[Head]
```

```
f[Sin[Cos]]
```

Evaluating now `f[Head[Head]]` yields `Symbol[Symbol]`.

```
f[Head[Head]]
```

The message is generated because the function `Symbol` expects a string as its argument.

```
??Symbol
```

For comparison, we also have the following.

```
Head[Head]
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

e) In this case, `Symbol` is not localized. It is interpreted as the built-in command. This behavior could not be easily predicted. This is an unexpected limitation. The lesson here is that no built-in commands should be used as pattern variables in function definitions. In addition to being dangerous, it also makes programs more difficult to read.

```
f[Symbol_Symbol] := Symbol
```

```
f[Sin]
```

```
f[x]
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

f) Here, the localization works again.

```
f[Integer_] := Integer
```

```
f[3]
```

```
f[x]
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

g) We first look at the `FullForm` of such an expression.

```
Hold[clock[#]&[#]&[#]&[#]&[hand]] // FullForm
```

The argument `hand` is passed from pure function to pure function. In the next input, each time the `Print[i]` is called in addition, and so the numbers 1 to 4 are printed.

```
clock[Print[4]; #]&[Print[3]; #]&[Print[2]; #]&[Print[1]; #]&[hand]
```

Just `clock[#]&[#]&[#]&[#]&[hand]` produces the same result.

```
clock[#]&[#]&[#]&[#]&[hand]
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

h) Here no first argument exists.

```
#1&[]
```

Here no second argument exists.

```
#2&[0]
```

`#0` reproduces the pure function itself, independent of the argument.

```
#0&[]
```

Again, no argument is here. The “empty argument sequence”, meaning `Sequence[]`, is returned.

```
##&[]
```

`##0` is not a defined expression. It is parsed as `SlotSequence[0]`, but no internal meaning has been defined for it.

```
##0&[] // Hold // FullForm
```

```
##0&[]
```

`###&[2, 3, 4]` gives the following result.

```
###&[2, 3, 4]
```

To understand the result better, we look at the `FullForm`.

```
FullForm[Hold[###&[2, 3, 4]]]
```

This result means `###` is interpreted as `Times[SlotSequence[1], Slot[1]]` and that the result is $(2\times3\times4)\times2 = 48$.

```
###&[2, 3, 4]
```

`a&b&c&[d]` yields the following result.

```
a&b&c&[d]
```

This result is because `a&b&c&` is a function whose second argument is a product of another function `Function[`⋱ `Times[Function[a], b]]`, and because the variable `c` is to be replaced by `d`.

```
a&b&c& // FullForm
```

```
(#&)&[2][3]
```

The result of the first operation (which does not depend on its arguments) is a pure function giving the value 2 for the argument 2.

```
(#&)&[2]
```

```
(#&)&[#]
```

```
(#&)&["CompleteGarbage"]
```

Without the round brackets (parentheses), the above expression would not make sense syntactically.

```
#&&[2]
```

Now, for the last example, evaluation proceeds from the inside out, and the inner `Slot` gives the 3.

```
(((#&)&)&)[1][2][3] // Hold // FullForm
```

```
(((#&)&)&)[1][2][3]
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

i) We first look at the result.

```
Evaluate[D[#, x]]&
```

`Evaluate` forces the computation of the inner expression.

```
D[#, x]
```

The result of this differentiation of `Slot[1]` with respect to `x` is 0, because `x` does not appear at all in `Slot[1]`. Hence, we get the following result.

```
0&
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

j) This unusual function definition is tailored for arguments that are typically arguments on the left-hand sides of function definitions, namely, those with head `Pattern`.

```
f[x_Pattern] := x^2
```

Thus, the right-hand side is computed to be `(x_Integer)^2`.

```
f[x_Integer]
```

Because `Set` (=) has the attribute `HoldFirst`, the left-hand side is not affected by the above function definition.

```
f[x_Integer] = f[x_Integer]
```

At the moment, `f` is defined as follows.

```
??f
```

Thus, `f[3]` is not an especially meaningful expression.

```
f[3]
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

k) `SetDelayed` has the attribute `HoldAll`. The `Evaluate` on the right-hand side of the following input disables `HoldAll` at the time the second argument of `SetDelayed` is evaluated.

```
Remove[f]
f[x_] := Evaluate[Expand[x]]
```

We now have the following definition of `f`.

```
??f
```

Thus, nothing is multiplied out in the following input.

```
f[(x + y)^3]
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

l) First, we look at the result.

```
2 // ((1 ~ #1 + #2& ~ #1)& @ #1)&
```

At first glance, this input appears somewhat cryptic because different forms of *Mathematica* functions and pure functions are mixed. Everything is a little clearer in the `FullForm`.

```
FullForm[Hold[2 // ((1 ~ #1 + #2& ~ #1)& @ #1)&]]
```

We now look in detail at the steps of the calculation. The computation begins with the application of `((1 ~ #1 + #2& ~ #1)& @ #1)&` to the argument 2 in the postfix form. The result is `(1 ~ #1 + #2& ~ #1)& @ 2`. Next, `(1 ~ #1 + #2& ~ #1)&` in the prefix form is applied, giving `1 ~ #1 + #2& ~ 2`. Finally, `#1 + #2&` in the infix form is applied to the two arguments 1 and 2, and we obtain the result 3.

```
2 // ((1 ~ #1 + #2& ~ #1)& @ #1)&
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

m) The `a` (which has the value `x_y`) is computed as the argument on the left-hand side of the function definition of `f`, resulting in the function definition `f[x_y] := x^2`. No definition exists for the symbol `a`.

```
a = x_y; f[a] := x^2;

??f
```

Thus, `f[a]` remains unevaluated.

```
f[a]
```

The definition matches an argument of the form `y[`*arguments*`]`.

```
f[y[a, a]]
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

n) The contents of this input are not particularly mathematically meaningful, but from a syntactic standpoint, it is allowed. `N` takes effect on all 1s, and gives `1.[1.]` each time, and these two factors are combined to a square.

```
N[1[1]] N[1[1]]
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

o) Here are the inputs of the function definitions of `f`.

```
f[x_Blank] := x^2;
f[x_y[x]] := x^-2;
f[x_(_y)] := x
f[Pattern[x, Blank[Blank[y]]]] := x^-1
```

We first look at the results of the individual summands. Here, the first definition of `f` applies.

```
f[_]
```

For y[z] as an argument, none of the above rules apply.

```
f[y[z]]
```

However, for y[z][x], they do apply, because the second definition above requires an argument with Head y[x].

```
x_y[x] // FullForm
```

In this case, y is the head of y[z] and x is the required argument of the head.

```
f[y[z][x]]
```

_head is Blank[head], that is, it has the head Blank, and the first of the above definitions applies.

```
f[_head]
```

The next one is analogous to the last summand, but it is obvious that the first definition applies because the argument is given in the FullForm.

```
f[Blank[y]]
```

The fourth definition requires an argument with the head Blank[y], which is the case for Blank[y][1].

```
f[Blank[y][1]]
```

The third definition of f applies to the last summand. The argument has to be a product of something and something with head y.

```
f[2 y[5]]
```

This form be clear if we look at the FullForm of the pattern.

```
x_(_y) // FullForm
```

It requires a product of two terms; something named x with something with the head y. Thus, after an appropriate reordering of the summands, we get the following result.

```
f[_] + f[y[z]] + f[y[z][x]] + f[_head] +
f[Blank[y]] + f[Blank[y][1]] + f[2 y[5]]
```

Σ (* session summary *) **TMGBs`PrintSessionSummary[]**

p) This expression remains completely unchanged (more correctly phrased: it undergoes the complete evaluation procedure, but returns unchanged).

```
Flat[Flat[Flat, Flat]]
```

This result is because Flat does not have the attribute Flat.

```
Attributes[Flat]
```

Here, Flat has the attribute Flat.

```
SetAttributes[Flat, Flat]
```

Thus, the above expression would simplify to the following.

```
Flat[Flat[Flat, Flat]]

ClearAttributes[Flat, Flat]
```

Σ (* session summary *) **TMGBs`PrintSessionSummary[]**

q) In the evaluation of v + a, the computation of v erases the value of a.

```
v := (Remove[a]; 1); a = 2;
v + a
```

Here we use the value of a before it is erased in the evaluation of v.

```
Remove[a, v];
v := (Remove[a]; 1);
a = 2;
a + v
```

Using v + a instead of a + v gives a different result.

```
Remove[a, v];
v := (Remove[a]; 1);
a = 2;
v + a
```

To understand the different results of the last two examples we observe that first, the arguments a and v are evaluated and then the Orderless attribute of Plus goes into effect.

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

r) The result is not 0 because Function[x, x] is identical with Function[y, y] from the standpoint of content, but not programming, because they have different variables.

```
Function[x, x] - Function[y, y]

Function[x, x] - Function[x, x]
```

Now let us look at the second input. The inner Function is here the dummy variable of the outer Function. The dummy variable of the outer Function appears in the body of the pure function in #^2& (=Function[Power[Slot[1], 2]]).

```
Function[Function, #^2&][Depth]
```

Applying the outer pure function to the argument Depth yields Depth[Slot[1]^2]. Freezing the body of the pure function after argument substitution, but before evaluation, shows this.

```
Function[Function, Hold[#^2&]][Depth]
```

Depth[Slot[1]^2] then results in 3.

```
Depth[Slot[1]^2]
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

s) During the evaluation of the definition of f with the initial argument 2, a definition for f corresponding to arguments with the head Integer is generated. This new, specialized definition is then used later in the calculation of f[2].

```
f[x_] := (f[Evaluate[Pattern[y, Blank[Head[x]]]]] :=
                              y + Head[x]; f[x^2])
f[2]
```

Here are all current definitions for f.

```
?f
```

To understand the computation, we can examine (by adding a Print statement) the DownValues of f on the fly when f is called.

```
Remove[f, y]

f[x_] := (Print["DownValues[f] beforehand:", DownValues[f]];
          f[Evaluate[Pattern[y, Blank[Head[x]]]]] := y + Head[x];
          Print["DownValues[f] subsequently:", DownValues[f]];
          f[x^2])
f[2]
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

t) This definition gives an infinite iteration because after the definition is applied, the result is again in a form in which the definition matches.

```
f[x_][y_] := f[y][x]

f[1][2]
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

u) The result in both cases is $11\,s + 111\,s^{11}$; because `s` from the argument and `s` from `Function` are different and so are treated independently (the `s` from the `Function` is a dummy variable and will be screened). The same independence holds for the two functions `functionArg` and `functionBody` from the second example.

```
Function[functionBody, Function[s,
                       functionBody][3]][11 s + 111 s^11]

Function[{functionArg, functionBody}, Function[functionArg,
                       functionBody][3]][s, 11 s + 111 s^11]
```

We can see inside the evaluation by wrapping the function `Hold` around the inner `Function`.

```
Function[functionBody, Hold @ Function[s,
                       functionBody][3]][11 s + 111 s^11]

Function[{functionArg, functionBody}, Hold @ Function[functionArg,
                       functionBody][3]][s, 11 s + 111 s^11]
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

v) The result is the number 9. The argument `Slot` is substituted for `f` inside the outer `Function`. The result is the pure function `# # &` with argument `3`. After its evaluation, we get 9.

```
Function[f, (# f)&[3]][#]
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

w) Looking at the `FullForm`, we clearly see the nesting of pure functions.

```
FullForm[Hold[# & & & & & & [1][2][3][4][5][6]]]
```

In the five outer pure functions, no reference is made to their arguments via `Slot`; the given arguments `1`, `2`, `3`, `4`, and `5` are irrelevant, and only the last (innermost) pure function is of the form `#&`, which means this innermost function, just gives its argument as the result of its application. The evaluation of the whole expression starts with evaluating the first (outermost) pure function, and the result is `# & & & & & [2][3][4][5][6]`. Then, the second pure function is evaluated, and so on. Finally, the last (innermost) pure function applied to `6` gives the result 6.

```
# & & & & & & [1][2][3][4][5][6]
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

x) This example is a refined remake of the function `noGo` from Subsection 3.1.1. In the first version, because of `Unevaluated`, the `x` on the right-hand side of the definition of `noGo` is replaced by `myNewVar` and not by the value of `myNewVar`. So, no error message is generated, and it evaluates.

```
noGo[x_] := (x = 11)
myNewVar = 1;
noGo[Unevaluated[myNewVar]]

myNewVar
```

`myNewVar` has been successfully assigned the value 11. The `HoldFirst` attribute has the same result. Again, the symbol `myNewVar` is plugged into `x = 11`, and `Set` can do the assignment.

```
SetAttributes[noGo, HoldFirst]
noGo[x_] := (x = 11)
myNewVar = 1;
noGo[myNewVar]

myNewVar
```

One more `Unevaluated` does not change anything in this case.

```
myNewVar = 1;
noGo[Unevaluated[myNewVar]]

myNewVar
```

But `Evaluate` wrapped around `myNewVar` forces `myNewVar` to evaluate to 11, despite the `HoldFirst` attribute, and no assignment can take place.

```
myNewVar = 1;
noGo[Evaluate[myNewVar]]

myNewVar
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

y) Here, the first expression is calculated.

```
Function[c, Slot[c] SlotSequence[c]&[1, 2, 3], Listable][{1, 2, 3}]
```

Because of the `Listable` attribute, the above expression is equivalent to the following expression.

```
{Slot[1] SlotSequence[1]&[1, 2, 3],
 Slot[2] SlotSequence[2]&[1, 2, 3],
 Slot[3] SlotSequence[3]&[1, 2, 3]}
```

Taking into account the meaning of #*i* and ##*i*, these reduce to the following products.

```
{1 1 2 3, 2 2 3, 3 3}
```

In the second input `Function[Slot, Slot[1]][C]` obviously evaluates to `C[1]`. `Function[Slot, Slot[1]]&[C]` evaluates to `Function[Slot,C]`. This pure function gets applied to the argument `1` and we get `C`. Finally, `C` gets applied to the argument `1` and the two `C[1]` cancel to yield `0`.

```
Function[Slot, Slot[1]]&[C][1][1] - Function[Slot, Slot[1]][C]
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

z) Here, the expression is evaluated.

```
Slot[1/2 + 1/2]&[1, 2]
```

`Function` has the attribute `HoldAll`, so the argument of `Slot[1/2 + 1/2]` is not evaluated to 1. But `Slot` needs a nonnegative integer as its argument, so an error message is generated and the expression remains unchanged. Forcing the evaluation of `1/2 + 1/2` with `Evaluate` gives the result 1.

```
Evaluate[Slot[1/2 + 1/2]]&[1, 2]
```

An `Evaluate` inside the `Slot` has, of course, no effect.

```
Slot[Evaluate[1/2 + 1/2]]&[1, 2]
```

The second expression gives `-2 #1 + #2` `#1` and `#2` parses as `Slot[1]` and `Slot[2]`. `#k` and `#`*k* parse as `Times[k, Slot[1]]` and `Times[`*k*`, Slot[2]]`. Afterwards, the variables `k` and *k* evaluate to their values `1` and `2`. This means the first expression `(#1 - #k)` evaluates to `0` and the second `(#2 - #`*k*`)` evaluates to `-2 #1 + #2 #1`, which is the result returned.

```
k = 1; k = 2; (#1 - #k) + (#2 - #k)
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

2. $(a+b)^{2n+1}$, Laguerre Polynomials

a) Here is the implementation of the notation. Note the input of the pattern, as well as the order of the factorization and multiplication.

```
niceForm[(a_Symbol + b_Symbol)^n_Integer] :=
                a^n + b^n + Factor[Expand[(a + b)^n] - a^n - b^n]
```

Now, we test if we still get the nice form for higher integers.

```
niceForm[(a + b)^3]
```

```
niceForm[(a + b)^5]
```

```
niceForm[(a + b)^7]
```

```
niceForm[(a + b)^9]
```

As we see, the pattern does not continue.

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

b) Using the series representation for the exponential function, we have the following identities:

$$\exp\left(-\frac{\partial}{\partial z}\, z\, \frac{\partial}{\partial z} + a\, \frac{\partial}{\partial z}\right) z^n = \sum_{k=0}^{\infty} \frac{1}{k\,!}\left(-\frac{\partial}{\partial z}\, z\, \frac{\partial}{\partial z} + a\, \frac{\partial}{\partial z}\right)^k z^n = \sum_{k=0}^{n} \frac{1}{k\,!}\left(-\frac{\partial}{\partial z}\, z\, \frac{\partial}{\partial z} + a\, \frac{\partial}{\partial z}\right)^k z^n.$$

The last formula is straightforward to implement. We use a pure function for the differential operator $(-\partial/\partial z\,(z\,\partial./\partial z) + a\,\partial./\partial z)$ and use `Nest` to realize it powers.

```
laguerreL[n_, a_, z_] := Factor[(-1)^n 1/n! *
          Sum[1/k! Nest[(-D[z D[#, z], z] + a D[#, z])&, z^n, k],
              {k, 0, n}]]
```

Here are the first few Laguerre polynomials. To evaluate the first five polynomials at once, we give `laguerreL` the attribute `Listable`.

```
SetAttributes[laguerreL, Listable]
```

```
laguerreL[{1, 2, 3, 4, 5}, a, z]
```

The so-calculated polynomials agree with the corresponding built-in ones.

```
LaguerreL[{1, 2, 3, 4, 5}, a, z]
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

3. $\frac{d}{d\,a\,x}\int^{a\,x} f(y)\,dy$

We look at an example.

```
f[x_] := Integrate[x^x, x]
```

```
D[f[s], s]
```

So far, everything is as expected. But now we try the following example.

```
D[f[s 𝕤], s 𝕤]
```

To get the desired result, we need a modified version of `Integrate`.

```
Unprotect[Integrate];
Integrate /: D[Integrate[int_, a_], a_] := int;
Protect[Integrate];
```

This input works as desired.

```
D[f[s 𝕤], s 𝕤]
```

As expected, the error message `Integrate::ilim` is generated because the iterator (i.e., the integration variable) does not have the form of a single variable, as required by `Integrate`, but is instead the product of two variables. Even if *Mathematica* could find the integral, the rule would not work because no integration is performed in the case of a product integration variable.

```
Integrate[Sin[y 𝕪], y 𝕪]
```

```
D[%, y 𝕪]
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

4. `Pattern[name, _]`

a) Here, the requirement that the *x* in `Pattern[`*x*`, _]` must be a symbol is not fulfilled.

```
Φ[Pattern[2, _]] = 2^2;
```

```
??Φ
```

The function definition from above applies to the third item in the following list because in this case, we have a suitable pattern, namely, `Pattern[2, _]` literally.

```
{Φ[2], Φ[3], Φ[Pattern[2, 2]], Φ[Pattern[2, _]]}
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

b) `I` is indeed a symbol, and syntactically everything is correct. However, in this case, `I` should not be used on the left.

```
Φ[Pattern[I, _]] = I^2;
```

```
??Φ
```

```
{Φ[2], Φ[I]}
```

Because of the `HoldFirst` attribute of `Pattern`, the variable `I` does not evaluate to `Complex[0, 1]`.

```
Pattern[I, _] // FullForm
```

```
pattern[I, _] // FullForm
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

c) Here, even the desired definition works, but this is not the right way to program. We should not use built-in symbols as names for a pattern.

```
Φ[Pattern[I, _]] := I^2;
```

```
??Φ
```

```
{Φ[2], Φ[I], Φ[Pattern[I, _]]}
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

d) `a[2]` has the head `a`, and it is not a `Symbol`.

```
Φ[Pattern[a[2], _]] := a[2]^2;
```

```
??Φ
```

```
{Φ[2], Φ[Pattern[a[2], _]]}
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

e) The `FullForm` of _ is the following.

```
FullForm[_]
```

Therefore, we naturally have the following for the `FullForm` of `_[_]` (`Blank[]` with argument `Blank[]`).

```
FullForm[_[_]]
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

5. Puzzles

a) The trick is to use `UpSetDelayed` to associate the result `d` with `Head`.

```
Head[b[c]] ^= d
```

```
b[c]
```

```
Head[b[c]]
```

We remove this special definition for b.

```
Clear[b]
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

b) We simply modify the rule for the computation of 2^2 and restart to reproduce exactly what was sought. (To have only one input for unprotecting `Power` and adding the new rule to it, we enclose both statements in parentheses.)

```
(Unprotect[Power]; Power[2, 2] = 2^4;)
```

```
Remove[f, x]
```

```
f[x_] := x^2
```

```
f[2]
```

We remove this rule to not interfere with later computations.

```
Power[2, 2] =.
```

```
Protect[Power];
```

```
2^2
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

c) We get a different result when the result of *expr*`[[1]]` evaluates nontrivially, say, for example, in `Hold[1 + 1]`.

```
Hold[1 + 1][[1, 1]]

Hold[1 + 1][[1]][[1]]
```

We also get different results in case *expr* in *expr*`[[1, 1]]` is a symbol and does not have a value, when it is not possible to extract the first part of the first part of *expr* because *expr* has depth 0.

```
iAmANewSymbolWithoutAValue[[1, 1]]
```

But taking the first part of the expression `Part[iAmANewSymbolWithoutAValue, 1]` (which has depth 1) just gives `iAmANewSymbolWithoutAValue`.

```
iAmANewSymbolWithoutAValue[[1]][[1]]

Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

d) This is the definition.

```
f[x_Real] := x^2
```

We can "fake" the head `Real` by wrapping `Real` around an arbitrary argument.

```
f[Real["1"]]

Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

e) For every ordinary *Mathematica* expression, this ends up with `Symbol`.

```
FixedPoint[Head, waikaki[2]]

FixedPoint[Head, 4.5]

FixedPoint[Head, "x[2]"]
```

Without a second argument in `FixedPoint`, we have another result. We can simulate this case of no second argument by using `Sequence[]`.

```
FixedPoint[Head, Sequence[]]

Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

f) Let us run the three examples.

```
Function[{s}, OIOI[s]][Unevaluated[κ]]

Function[{s}, OIOI[s]][Unevaluated[Unevaluated[κ]]]

Function[{s}, OIOI[s]][Unevaluated[
                          Unevaluated[Unevaluated[κ]]]]
```

Wrapping `Unevaluated` around an expression gives the expression in unevaluated form to the outer function, which means that in the first example κ is given to `OIOI` and `OIOI[κ]` is returned from the `Function`.

In the second case, `Unevaluated[κ]` is substituted for `s` inside the function. As a result, `OIOI` is called with an argument with head `Unevaluated`, and κ is passed unevaluated to `OIOI`. The result is again `OIOI[κ]`.

In the third example, `Unevaluated[Unevaluated[κ]]` is given to `OIOI`, the outer `Unevaluated` is stripped away, and the result is `OIOI[Unevaluated[κ]]`.

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

g) If `Infinity` is input, it is converted to `DirectedInfinity[1]`, as we can see in `FullForm`.

```
Infinity // FullForm
```

Extracting the first element of `DirectedInfinity[1]` gives 1.

```
Infinity[[1]]
```

This 1 is used as an argument of `DirectedInfinity`, which in this case has the output form `Infinity`.

```
DirectedInfinity[1] // OutputForm
```

Because the output form and the internal form differ from each other, we can extract the first part.

```
%[[1]]
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

h) The output of the three inputs `a^2`, `Clear[a]`, and `Remove[a]` shows that some additional rules have been given. Two possible ways to achieve the outputs shown are to give additional rules to `Power`, `Clear`, and `Remove` or to use upvalues on `a`. Here, both ways are demonstrated. First, the built-in functions are unprotected and overloaded (we wrap parentheses around all pieces of the first input to avoid incrementing the `In` counter).

```
(Unprotect[{Power, Clear, Remove}];
 a^2 = b;
 Clear[a] = a;
 Remove[a] = a;)
```

```
a^2
```

The definition `a^2 = b` is stored as a downvalue for `Power`.

```
{UpValues[a], DownValues[Power]}
```

```
Clear[a]
```

```
Remove[a]
```

Using `Remove` with the argument `"a"` removes all of the above definitions.

```
Remove["a"]
```

```
Clear[Power]
```

```
??a
```

Now, we use upvalues on `a`.

```
(a /: Clear[a] = a;
 a /: Remove[a] = a;
 a /: a^2 = b;)
```

```
a^2
```

Now the definition is stored as an upvalue for `a`.

```
{UpValues[a], DownValues[Power]}
```

```
Clear[a]
```

```
Remove[a]
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

i) The result of evaluating the input will be 6.

```
f[_] := (f[_] := #0[# + 1]; # + 1)&[1]
f[f[f[f[f[2]]]]]
```

To see why the result is 6, let us analyze the first application of `f` to the argument `2`. When `f` gets called with an arbitrary argument (the _ in the left-hand side of the definition of `f`), the right-hand side will be evaluated. The right-hand has the structure of a pure function that is applied to the argument 1. The body of the pure function is `(f[_] := #0[# + 1]; # + 1)`. This means that by evaluating the body a new definition for `f`, namely the old one with a new argument for the pure function, is generated. The result of evaluating of `f[_]` will be the argument of the pure function on the right-hand side + 1.

```
f[_] := (f[_] := #0[# + 1]; # + 1)&[1]
f[2]
```

Here is the current definition of `f`. We see that the argument of the pure function is now `1 + 1`.

```
DownValues[f]
```

In the next application of `f`, the above procedure is carried out again and we end up with the argument `2 + 1` of the pure function. (The above 1 + 1 was evaluated in the argument, but the 2 + 1 stays unevaluated because it is on the right-hand side of a `SetDelayed` statement.)

```
f[2]
```

```
DownValues[f]
```

So after applying `f` five times to the 2, we have the result 6.

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

6. Different Patterns

a) By looking at the `FullForm` of the left-hand sides of the function definitions, we recognize the coded pattern.

```
f[x_, a[b_, c_]_] // FullForm
```

```
f[x_, a[b_, c_]_] := 𝕗[x, a, b, c]
```

An arbitrary function with two arguments as a second argument was not coded.

```
f[x, a[y, z]]
```

Instead, the second argument of `f` must be a product of `a[`*twoArguments*`]` and *something*. Here are two inputs that match this pattern.

```
f[x, a[y, z] b]
```

```
f[x_, a[b_, c_]_]
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

b) Now, the second argument of `f` is an arbitrary function of two arbitrary arguments.

```
f[x_, a_[b_, c_]] // FullForm
```

```
f[x_, a_[b_, c_]] := 𝕗[x, a, b, c]
```

Now, `f[x, a[y, z]]` evaluates nontrivially.

```
f[x, a[y, z]]
```

The pattern also matches the following two inputs.

```
f[x, a[𝕓[𝕕, 𝕖], 𝕗[𝕘, 𝕙]]]
```

```
f[x, Plus[y, z]]
```

Here, the second argument of `f` is not an object with two arguments at the time the definition of `f` goes into effect, but it is evaluated before to 3.

```
f[x, Plus[1, 2]]
```

Avoiding the evaluation of `Plus[1, 2]` yields a nontrivial result.

```
f[x, Unevaluated[Plus[1, 2]]]
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

c) `f` is now a function of one argument that is a product of three arbitrary expressions.

```
f[a_ b_ c_] // FullForm

f[a_ b_ c_] := 𝕗[a, b, c]

f[b c a]
```

In principle, all three expressions can be the same. But in the following case, `a a a` is combined to `a^3`, and the result is only one argument with head `Power`.

```
f[a a a]
```

Analogous to case b), this does not work.

```
f[1 2 3]
```

`Unevaluated` avoids that the arguments of `f` are evaluated before `f` deals with them.

```
f[Unevaluated[1 2 3]]
```

What if we had given `f` the attribute `HoldAll` (or `HoldFirst` or `HoldRest`)?

```
Remove[f, 𝕗, x, y, z, a, b, c]

SetAttributes[f, HoldAll]

f[a_ b_ c_] := 𝕗[a, b, c]
```

Then, `a a a` would not have been replaced by `a^3`, and the definition of `f` would have been applied.

```
f[a a a]
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

d) Here, the pattern _ _ on the left-hand side is evaluated to `_^2` before the actual definition of the function is carried out.

```
f[_ _] // FullForm

f[_ _] := 𝕗
```

Thus, the resulting function definition of `f` only fits an argument that is a square.

```
f[a b]

f[a a]
```

To encode a product of two distinct factors in this case, we would for instance have had to use the following input.

```
Remove[f, a, b]

f[a_ b_] // FullForm
```

```
f[a_ b_] := 𝕗
```

```
f[a b]
```

The function definition no longer fits for `a^2` now; we have only one argument with head `Power`.

```
f[a a]
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

7. `Plot[`*numberFunction*`]`

Here are the function definitions.

```
f1[x_Integer] = 3;
f1[x_Rational] = 2;
f1[x_Real] = 1;
```

Here is an attempt to plot `f1`.

```
Plot[f1[x], {x, -3, 3}];
```

It fails because `Plot` used only machine numbers for plotting. Functions are first compiled to speed up the computation of their function values before plotting. (We return to the issue of compilation in great detail in Chapter 1 of the Numerics volume [63✱].) `Plot` supplies real values (head `Real`) to the functions, and the two definitions for `f2` do not match.

```
f2[x_Integer] = 3;
f2[x_Rational] = 2;
Plot[f2[x], {x, -3, 3}];
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

8. Tower of Powers

Here is the program for this power tower.

```
powerTower[number_, n_] := number^powerTower[number, n - 1];
powerTower[number_, 1] = number;
```

Here are the lowest levels.

```
powerTower[z, 2]
```

```
powerTower[z, 4]
```

```
powerTower[z, 8]
```

```
TreeForm[powerTower[z, 4]]
```

For $z = \sqrt{2}$, we get the following numerical results.

```
Table[powerTower[Sqrt[2.], n], {n, 1, 30}]
```

Based on these numerical result we conjecture that 2 is the solution. Physicists will recognize a Dyson equation in the power tower of the form: $x = number^x$. For $number = \sqrt{2}$, the solution for x is obviously 2. (For general *number* the solution is, modulo branch cuts, $x = -W(-\ln(number))/\ln(number)$. Here $W(z)$ is the `ProductLog` function; it will be discussed in the Symbolics volume [64✱].)

Here is a similar example: $\sqrt{2\sqrt{2+\sqrt{\cdots+\sqrt{2}}}}$.

```
FixedPointList[N[Sqrt[2 + #]]&, Sqrt[2]]
```

Using high-precision numbers, we can see the difference to 2.

```
N[2 - FixedPointList[Sqrt[2 + #]&, N[Sqrt[2], 30], 30]]
```

For more details on power towers, see [34✱], [38✱], [71✱], [2✱], [12✱], [7✱], [27✱], [18✱], [47✱], [8✱], [13✱], [52✱], [21✱], [53✱], and [5✱]; for generalizations, see [29✱], [57✱], [68✱].

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

9. Cayley Multiplication

We want to find a multiple product, but at each step know only the binary result in view of the associativity of the operation, thus we apply the attribute `Flat` at each step. Here are the implementations of the individual rules.

```
SetAttributes[𝒞𝒯, Flat]

𝒞𝒯[e, a] = a; 𝒞𝒯[a, e] = a; 𝒞𝒯[e, b] = b; 𝒞𝒯[b, e] = b;
𝒞𝒯[e, c] = c; 𝒞𝒯[c, e] = c; 𝒞𝒯[a, b] = c; 𝒞𝒯[b, a] = c;
𝒞𝒯[a, c] = b; 𝒞𝒯[c, a] = b; 𝒞𝒯[b, c] = a; 𝒞𝒯[c, b] = a;
𝒞𝒯[a, a] = e; 𝒞𝒯[e, e] = e; 𝒞𝒯[b, b] = e; 𝒞𝒯[c, c] = e;
```

The desired expression turns out to be the `e`.

```
𝒞𝒯[a, b, c, a, c, e, a, c, b, b, c, a, e, a, c, c, a, b, a, c,
   a, c, a, e, b, b, a, a, e, c, b, b, a, a, c, e, e, e, a, a,
   b, b, b, a, b, c, b, c, a, a, c, c, c, b, a, a, e, e, c]
```

The same result can be obtained by applying 𝒞𝒯 to two arguments repeatedly. To achieve this, we have to remove the attribute `Flat` from 𝒞𝒯.

```
ClearAttributes[𝒞𝒯, Flat]
```

Because of the associativity, we can group things in many different ways, for example, as in the following.

```
𝒞𝒯[𝒞𝒯[𝒞𝒯[𝒞𝒯[𝒞𝒯[𝒞𝒯[𝒞𝒯[𝒞𝒯[𝒞𝒯[𝒞𝒯[𝒞𝒯[𝒞𝒯[𝒞𝒯[𝒞𝒯[𝒞𝒯[𝒞𝒯[
𝒞𝒯[𝒞𝒯[𝒞𝒯[𝒞𝒯[𝒞𝒯[𝒞𝒯[𝒞𝒯[𝒞𝒯[𝒞𝒯[𝒞𝒯[𝒞𝒯[𝒞𝒯[𝒞𝒯[𝒞𝒯[𝒞𝒯[𝒞𝒯[
𝒞𝒯[𝒞𝒯[𝒞𝒯[𝒞𝒯[𝒞𝒯[𝒞𝒯[𝒞𝒯[𝒞𝒯[𝒞𝒯[𝒞𝒯[𝒞𝒯[𝒞𝒯[𝒞𝒯[𝒞𝒯[𝒞𝒯[𝒞𝒯[
𝒞𝒯[𝒞𝒯[𝒞𝒯[𝒞𝒯[𝒞𝒯[𝒞𝒯[𝒞𝒯[𝒞𝒯[𝒞𝒯[𝒞𝒯[𝒞𝒯[a, b], c], a], c],
e], a], c], b], b], c], a], e], a], c], c], a], b], a], c], a], c],
a], e], b], b], a], a], e], c], b], b], a], a], c], e], e], e], a],
a], e], b], b], b], a], b], c], b], c], a], a], c], c], c], b], a],
a], e], e], c]
```

Here is another example.

```
𝒞𝒯[𝒞𝒯[𝒞𝒯[𝒞𝒯[𝒞𝒯[𝒞𝒯[a, b], 𝒞𝒯[c, a]], 𝒞𝒯[𝒞𝒯[c, e], 𝒞𝒯[a, c]]],
𝒞𝒯[𝒞𝒯[𝒞𝒯[b, b], 𝒞𝒯[c, a]], 𝒞𝒯[𝒞𝒯[e, a], 𝒞𝒯[c, c]]]], 𝒞𝒯[𝒞𝒯[
𝒞𝒯[𝒞𝒯[a, b], 𝒞𝒯[a, c]], 𝒞𝒯[𝒞𝒯[a, c], 𝒞𝒯[a, e]]], 𝒞𝒯[𝒞𝒯[𝒞𝒯[b, b],
𝒞𝒯[a, a]], 𝒞𝒯[𝒞𝒯[e, c], 𝒞𝒯[b, b]]]]], 𝒞𝒯[𝒞𝒯[𝒞𝒯[𝒞𝒯[𝒞𝒯[𝒞𝒯[a, a],
𝒞𝒯[c, e]], 𝒞𝒯[𝒞𝒯[e, e], 𝒞𝒯[a, a]]], 𝒞𝒯[𝒞𝒯[𝒞𝒯[e, b], 𝒞𝒯[b, b]],
𝒞𝒯[𝒞𝒯[a, b], 𝒞𝒯[c, b]]]], 𝒞𝒯[𝒞𝒯[𝒞𝒯[c, a], 𝒞𝒯[a, c]], 𝒞𝒯[𝒞𝒯[c, c],
𝒞𝒯[b, a]]]], 𝒞𝒯[𝒞𝒯[a, e], 𝒞𝒯[e, c]]]]
```

If we want to know how often various rules were applied, we can count each application by incrementing a counter.

```
Remove[CT];

SetAttributes[CT, Flat]

initializeCounter :=
(count[ 1] = 0; count[ 2] = 0; count[ 3] = 0;
 count[ 4] = 0; count[ 5] = 0; count[ 6] = 0;
 count[ 7] = 0; count[ 8] = 0; count[ 9] = 0;
 count[10] = 0; count[11] = 0; count[12] = 0;
 count[13] = 0; count[14] = 0; count[15] = 0;
 count[16] = 0;)

CT[e, a] := (count[ 1] = count[ 1] + 1; a);
CT[a, e] := (count[ 2] = count[ 2] + 1; a);
CT[e, b] := (count[ 3] = count[ 3] + 1; b);
CT[b, e] := (count[ 4] = count[ 4] + 1; b);
CT[e, c] := (count[ 5] = count[ 5] + 1; c);
CT[c, e] := (count[ 6] = count[ 6] + 1; c);
CT[a, b] := (count[ 7] = count[ 7] + 1; c);
CT[b, a] := (count[ 8] = count[ 8] + 1; c);
CT[a, c] := (count[ 9] = count[ 9] + 1; b);
CT[c, a] := (count[10] = count[10] + 1; b);
CT[b, c] := (count[11] = count[11] + 1; a);
CT[c, b] := (count[12] = count[12] + 1; a);
CT[a, a] := (count[13] = count[13] + 1; e);
CT[e, e] := (count[14] = count[14] + 1; e);
CT[b, b] := (count[15] = count[15] + 1; e);
CT[c, c] := (count[16] = count[16] + 1; e);

initializeCounter

CT[a, b, c, a, c, e, a, c, b, b, c, a, e, a, c, c, a, b, a, c,
   a, c, a, e, b, b, a, a, e, c, b, b, a, a, c, e, e, e, a, a,
   b, b, b, a, b, c, b, c, a, a, c, c, c, b, a, a, e, e, c]
```

Here is the number of applications for each of the rules.

```
??count

initializeCounter

CT[CT[CT[CT[CT[CT[a, b], CT[c, a]], CT[CT[c, e], CT[a, c]]],
CT[CT[CT[b, b], CT[c, a]], CT[CT[e, a], CT[c, c]]]], CT[CT[
CT[CT[a, b], CT[a, c]], CT[CT[a, c], CT[a, e]]], CT[CT[CT[b, b],
CT[a, a]], CT[CT[e, c], CT[b, b]]]]], CT[CT[CT[CT[CT[CT[a, a],
CT[c, e]], CT[CT[e, e], CT[a, a]]], CT[CT[CT[e, b], CT[b, b]], CT[
CT[a, b], CT[c, b]]]], CT[CT[CT[c, a], CT[a, c]], CT[CT[c, c],
CT[b, a]]]], CT[CT[a, e], CT[e, c]]]]
```

Carrying out the multiplication in a different order results in a different result for the counters.

```
??count
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

References

✱1 J. Abott. *Math. Comput.* 71, 407 (2002). ftp://cocoa.dima.unige.it/papers/Abbott00.dvi

✱2 E. J. Allen. *Math. Gaz.* 69, 261 (1985).

✱3 T. M. Apostol. *Am. Math. Monthly* 107, 738 (2000).

✱4 J. M. Ash. *Math. Mag.* 69, 207 (1996).

✱5 E. Barbeau. *Coll. J. Math.* 25, 130 (1995).

✱6 F. C. Bauer (eds.). *Logic, Algebra and Computation*, NATO ASI F 79, Springer-Verlag, New York, 1991. *BookLink*

✱7 B. C. Berndt. *Ramanujan*'*s Notebooks* I, Springer-Verlag, New York, 1985. *BookLink*

✱8 B. C. Berndt, Y.-S. Choi, S.-Y. Kang in B. C. Berndt, F. Gesztesy (eds.). *Continued Fractions: From Analytic Number Theory to Constructive Approximation*, American Mathematical Society, Providence, 1999. *BookLink*

✱9 M. Bezem, J. F. Groote (eds.). *Typed Lambda Calculi and Applications*, Springer-Verlag, Berlin, 1993. *BookLink*

✱10 I. N. Bronshtein, K. A. Semandyayev. *Handbook of Mathematics*, Van Nostrand, New York, 1985. *BookLink (2)*

✱11 X. Buff, C. Henriksen. *Nonlinearity* 16, 989 (2003). *DOI-Link*

✱12 A. P. Bulanov. *Izvestiya RAN* 62, 901 (1998).

✱13 A. P. Bulanov. *Sbornik Math.* 192, 1589 (2001). *DOI-Link*

✱14 H. Canary, C. Edquist, S. Lachterman, B. Younger. *arXiv:math.CO*/0407115 (2004). *Get Preprint*

✱15 D. Coppersmith, J. Davenport. *Acta Arithm.* 58, 79 (1991).

✱16 R. M. Corless. *Math. Mag.* 71, 34 (1998).

✱17 G. Dattoli, H. M. Srivastava, C. Cesarano. *Appl. Math. Comput.* 124, 117 (2001). *DOI-Link*

✱18 S. M. Didukh, E. L. Pekarev. *MPS: Pure mathematics*/0205011 (2000). http://www.mathpreprints.com/math/Preprint/Pekarev/20020520/1/

✱19 T. Ehrhard, L. Regnier. *Theor. Comput. Sc.* 309, 1 (2003). *DOI-Link*

✱20 E. Eisenberg, A. Baram. *J. Phys.* A 33, 1729 (2000). *DOI-Link*

✱21 L. Gerber. *Proc. Am. Math. Soc.* 41, 205 (1973).

✶22 W. J. Gilbert. *Fractals* 9, 251 (2001). *DOI-Link*

✶23 J. W. Gray. *Categorial Semantics of Programming Languages*, Addison-Wesley, Redwood City, 1991.

✶24 I. Gumowski, C. Mira. *Recurrences and Discrete Dynamic Systems*, Springer-Verlag, Berlin, 1980. *BookLink*

✶25 C. Hankin. *Lambda Calculi*, Clarendon Press, Oxford, 1994. *BookLink (2)*

✶26 N. D. Hayes. *Quart. J. Math. Oxford* 3, 81 (1952).

✶27 A. Herschfeld. *Am. Math. Monthly* 42, 419 (1935).

✶28 H. Hirayama in P. Schiavone, C. Constanda, A. Mioduchowski (eds.). *Integral Methods in Science and Engineering*, Birkhäuser, Boston, 2002. *BookLink*

✶29 C. Horowitz. *Israel J. Math.* 29, 42 (1978).

✶30 J. Hubbard, D. Schleicher, S. Sutherland. *Invent. Math.* 146, 1 (2001). *DOI-Link*

✶31 M. Jeong, G. O. Kim, S.-A Kim. *Comput. Graphics* 26, 271 (2002). *DOI-Link*

✶32 W. P. Johnson. *Am. Math. Monthly* 109, 273 (2002).

✶33 K. Kneisl. *Chaos* 11, 359 (2001). *DOI-Link*

✶34 R. A. Knoebel. *Am. Math. Monthly* 88, 235 (1981).

✶35 D. E. Knuth in V. Lifschitz (ed.). *Artificial Intelligence and Mathematical Theory of Computation*, Academic Press, Boston, 1991. *BookLink*

✶36 T. Komatsu. *Fibon. Quart.* 39, 336 (2001).

✶37 J. L. Krivine. *Lambda Calculus, Types and Models*, Ellis Horwood, Masson, 1993. *BookLink*

✶38 D. Laugwitz. *Elem. Math.* 45, 89 (1990).

✶39 S. Lou, C. Chen, X. Tang. *J. Math. Phys.* 43, 4078 (2002). *DOI-Link*

✶40 Y. Y. Lu. *Appl. Num. Math.* 27, 141 (1998). *DOI-Link*

✶41 B. Martin in J. Landsdown, R. A. Earnshaw (eds.). *Computers in Art, Design and Animation*, Springer-Verlag, New York, 1989. *BookLink*

✶42 M. D. Meyerson. *Math. Mag.* 69, 198 (1996).

✶43 J. W. Neuberger. *Math. Intell.* 21, n3, 18 (1999).

✶44 A. Oberschelp. *Rekursionstheorie*, BI, Mannheim, 1993. *BookLink*

✶45 P. Odifreddi. *Classical Recursion Theory*, North Holland, Amsterdam, 1992. *BookLink (3)*

✶46 P. Odifreddi in C. S. Calude, M. J. Dinneen, S. Sburlan (eds.). *Combinatorics, Computability and Logic*, Springer-

Verlag, London, 2001. *BookLink*

✱47 C. S. Ogilvy. *Am. Math. Monthly* 77, 388 (1970).

✱48 B. J. Pierce in A. B. Tucker, Jr. (ed.). *The Computer Science and Engineering Handbook*, CRC Presss, Boca Raton, 1997. *BookLink (2)*

✱49 T. Prellberg. *arXiv:math.CO*/0005008 (2000). *Get Preprint*

✱50 T. Prellberg in F. Garvan, M. Ismail (eds.). *Symbolic Computation, Number Theory, Special Functions, Physics and Combinatorics*, Kluwer, Dordrecht, 2001. *BookLink*

✱51 G. E. Revesz. *Lambda-Calculus, Combinators and Functional Programming*, Cambridge University Press, Cambridge, 1986. *BookLink*

✱52 G. Schuske, W. J. Thron. *Proc. Am. Math. Soc.* 112, 527 (1962).

✱53 L. D. Servi. *Am. Math. Monthly* 110, 326 (2003).

✱54 H. Simmons. *Derivation and Computation*, Cambridge University Press, Cambridge, 2000. *BookLink*

✱55 P. H. Sterbenz, C. T. Fike. *Math. Comput.* 23, 313 (1969).

✱56 I. Stewart. *Sci. Am.* n12, 144 (1992).

✱57 G. Szekeres. *J. Austral. Math. Soc.* 2, 301 (1962).

✱58 J. Tamura in J. Akiyama, Y. Ito, S. Kanemitsu, T. Kano, T. Mitsui, I. Shiokawa (eds.). *Number Theory and Combinatorics*, World Scientific, Singapore, 1985. *BookLink*

✱59 X. Tang, S. Lou, Y. Zhang. *Phys. Rev.* E 66, 046601 (2002). *DOI-Link*

✱60 X. Tang, S. Lou. *arXiv:nlin.SI*/0210009 (2002). *Get Preprint*

✱61 B. A. Trakhtenbrot in R. Herken (ed.). *The Universal Turing Machine: A Half Century Later*, Kammerer & Unverzagt, Hamburg, 1988. *BookLink*

✱62 M. Trott. *The Mathematica GuideBook for Graphics*, Springer-Verlag, New York, 2004. *BookLink*

✱63 M. Trott. *The Mathematica GuideBook for Numerics*, Springer-Verlag, New York, 2005.

BookLink

✱64 M. Trott. *The Mathematica GuideBook for Symbolics*, Springer-Verlag, New York, 2005.

BookLink

✱65 J. van Benthem. *Language in Action*, North Holland, Amsterdam, 1991. *BookLink (2)*

✱66 I. Vardi. *Computational Recreations in Mathematica*, Addison-Wesley, Reading, 1991. *BookLink*

✱67 J. L. Varona. *Math. Intell.* 24, n1, 37 (2002).

✱68 P. Walker. *Math. Comput.* 57, 723 (1991).

✱69 G. Walz. *Asymptotics and Extrapolation*, Akademie Verlag, Berlin, 1996. *BookLink (2)*

✱70 S. R. Wassell. *Math. Mag.* 73, 111 (2000).

✱71 R. O. Weber, J. Roumeliotis. *Austral. Math. Soc. Gaz.* 22, 183 (1995).

✱72 E. Wingler. *Am. Math. Monthly* 97, 836 (1990).

✱73 A. Wünsche. *J. Comput. Appl. Math.* 133, 665 (2001). *DOI-Link*

✱74 L. Yau, A. Ben-Israel. *Am. Math. Monthly* 105, 806 (1998).

PROGRAMMING

CHAPTER 4

Meta-*Mathematica*

4.0 Remarks

The title of this chapter calls for some explanation. This chapter largely discusses functions and functionalities of *Mathematica* that are either unrelated or only indirectly related to mathematics and together with the former, the *Mathematica* purpose-defining tagline *Mathematica–A System for Doing Mathematics by Computer* this explains the title. This chapter does not deal with any “meta-mathematical” (in the sense of Gödel-Turing-Chaitin [3✱], [4✱], [5✱], [13✱], [12✱], [15✱], [7✱], [14✱], [6✱]) issues.

We begin this chapter with a discussion about on-line help within the *Mathematica* kernel (the use of the help browser within the front end should not need much explanation). We will discuss the storage and input of data and definitions and quickly go over debugging. Although important, we will not use debugging much in this book because all programs presented should work properly. Then, we go on to programming techniques (subprograms, variable protection, and contexts) and discuss the order in which transformations are performed on any *Mathematica* input. Despite its nonmathematical character, a knowledge of the material in Sections 4.6 and 4.7 is essential for the efficient use of *Mathematica*.

```
(* no spelling warnings, set fonts for tick labels, ... *)
Get[ToFileName[ReplacePart["FileName" /.
 NotebookInformation[EvaluationNotebook[]], "Initialization.m", 2]]];
```

4.1 Information on Commands

■ 4.1.1 Information on a Single Command

It is often useful to make a list of the names of all symbols that have already been introduced, for example, during a long *Mathematica* session. This can be accomplished with `?*`, but the resulting list cannot be further manipulated because it is not accessible through the output history `Out[]`.

`Information[`*whatWeAmInterestedIn*`]`
  or
`?`*whatWeAreInterestedIn*

  gives the most important information on the built-in *Mathematica* function *whatWeAreInterestedIn*. The output is not in the form `Out[`*i*`] = ` *info*. The usual string metacharacters `*` and `@` can be used to specify *whatWeAreInterestedIn*.

Here is the use of `Information`.

```
Information[Information]
```

The following two inputs show the different behaviors of `Information` and `?` on short forms of *Mathematica* operators.

```
Information[Plus]
```

```
? +
```

If we use a construction of the form `f[`arg_1`][`arg_2`] =` *something*, `Information` of the definition is associated with `f`, not with `f[`arg_1`]`. (In the following case as a subvalue for `myNestedFunction`.)

```
myNestedFunction[parameter][argument_] := parameter argument
```

```
??(myNestedFunction[parameter])
```

```
??myNestedFunction
```

Here are all commands beginning with `Ac` (to avoid a long output, we use the two starting letters).

```
?Ac*
```

To get a list (head `List`) of these symbols using *Mathematica*, we can use `Names`.

`Names["`*functionNameLetters*`"]`
gives a list (head `List`) of the names of already existing symbols that match *functionNameLetters*, taking into account metacharacters in *functionNameLetters*. (All names from all visible contexts, which are the ones in `$Path`, are listed.)

Using this command, it is possible to find out how many commands, attributes, and options are in *Mathematica* (provided we have not yet introduced any symbols of our own, which is the case in the present session). This list includes user-defined and internal variables and, if used in the form `Names["*`*"]`, all names from all contexts (see below). Here, we create a list that could be further manipulated in *Mathematica*, containing only those commands beginning with `A`.

```
Names["Ac*"]
```

Here is the total number of currently visible built-in commands.

```
Length[Names["*"]] -
(* subtract myNestedFunction, parameter, argument *) 3
```

Of these commands, about 125 begin with `$` rather than an uppercase letter, as is usual in *Mathematica*.

```
Length[Names["@*"]] - 3
```

```
Length[Names["$*"]]
```

We discuss some of these `$`*name* commands later in this chapter. Typically, some information and messages are associated with every command in *Mathematica*:

■ how to use the function (for a more detailed description, see the on-line *Mathematica* book in the help browser)

■ warning and error messages

The messages can be obtained using `Messages`.

`Messages[`*symbol*`]`
gives a list of all messages associated with the symbol.

Here are two examples of messages generated because of “incorrect” use of functions or because of “unexpected” arguments. Here, `Part` is called with a noninteger second argument.

```
Part[12.34 a^34, -23.56]
```

The following example is an incorrect attempt to plot the function $f(x) = x^2$. Although we discuss the details for graphics in Chapter 1 of the Graphics volume [27✱], it is immediately clear that a direct use of the English syntax is inappropriate because it will be interpreted as a product.

```
Plot[y(x) = x^2, between x = -1 and x = 1,
      (* some naive option settings *)
      blue background, red line,  thick green frame,
      big bold black label "The Quadratic" on top]
```

Because they need quite a bit of memory, messages are not automatically present in a *Mathematica* session. We can still get all messages by explicitly reading in the appropriate file. (In the following inputs, we will use a certain number of commands that have not yet been discussed; for now, the emphasis here is on the *Mathematica* output.)

The following reads in the file of all usage messages.

```
Get[ToFileName[
    {$TopDirectory, "SystemFiles", "Kernel", "TextResources",
                 $Language}, #]]& /@
  {(* usage messages *)  "Usage.m",
   (* warning and error messages *)  "Messages.m"};
```

Every message has its own name; we can get the message using `MessageName`.

`MessageName[`*symbol*`, "`*message*`"]`
or
symbol `::"`*message*`"`
gives the message *message* for the symbol *symbol.*

For example, here is the usage message of `SetAttributes`.

```
SetAttributes::"usage"
```

The most important symbol in connection with messages is `General`.

`General`
is the symbol associated with general system information and system messages.

Here are some of the general system messages.

```
Messages[General] // Short[#, 12]&
```

The total number of messages belonging to `General` is more than 200.

```
Length[%]
```

We now collect all messages in the list `allMessages`. (We concentrate on the result, not the programming of the following input.)

```
systemCommands = Names["System`*"];

(* clear the ReadProtected attribute *)
If[MemberQ[Attributes[#], ReadProtected],
   ClearAttributes[#, ReadProtected]]& /@
     Apply[Unevaluated, ToHeldExpression /@
              DeleteCases[systemCommands, "I"], {1}];

(* make list of all messages *)
allMessages = (Messages @@ #)& /@ (ToHeldExpression[#]& /@
                                   DeleteCases[systemCommands, "I"]);
```

Because of space limitations, we do not look at the list. `allMessages` contains nearly 3000 messages.

```
ℓ = Length[Flatten[allMessages]]
```

Here are five entries from the beginning, the middle, and the end of the list `allMessages`.

```
Take[Flatten[allMessages], {1, 5}]

Take[Flatten[allMessages], {ℓ - 5, ℓ}]
```

These entries take up a total of about 600 kB.

```
ByteCount[allMessages]
```

The following gives some idea of how many messages are associated with the various commands. (Look only at the result, not the programming.)

```
With[{cp = CellPrint[Cell[StringJoin[##], "PrintText"]]&},
Apply[
Which[(* write the various cases;
    ∘ stands again for Mathematica-generated text *)
      #1 === "1" && #2 === "1", cp[  (* 1 command, 1 message *)
          "∘ There is 1 system command with 1 message."],
      #1 === "1" && #2 =!= "1", cp[  (* 1 command, n messages *)
          "∘ There is 1 system command with ", #2, " messages."],
      #1 =!= "1" && #2 === "1", cp[  (* n commands, 1 message *)
          "∘ There are ", #1, " system commands with 1 message."],
      True,                      cp[  (* n commands, n messages *)
          "∘ There are ", #1, " system commands with ", #2,
                                                     " messages."]]&,
 (* the count *)
 Map[ToString, (Function[p, {Count[#, p], p}] /@ Union[#])&[
                         Length /@ allMessages], {-1}], {1}]];
```

Here (and earlier in the last two chapters), we have made use of `CellPrint`.

`CellPrint[`*cellExpression*`]`

prints the cell (head `Cell`) *cellExpression* as a cell into the currently selected notebook.

A simpler version of `CellPrint` that does not allow styling is `Print`.

`Print[`$expression_1$`, ` $expression_2$`, …, ` $expression_n$`]`

prints $expression_1$ up to $expression_n$ joined together.

Using one (or more) explicit newline characters as the arguments to `Print`, we can write a sequence of expressions to different lines.

```
Print[1, "\n", 2, "\n\n", "    and an indented 3."]
```

The messages have names, which are not complete words, as opposed to the *Mathematica* naming convention for commands (when programming our own messages, we can of course use longer, more descriptive namings). Here is a part of the complete list shown.

```
Union[(((Hold @@ #[[1]]) /. {MessageName -> List})[[1, 2]])& /@
                 Flatten[allMessages]] // OutputForm // Short[#, 6]&
```

This is the total number of such abbreviations.

```
Length[%]
```

Thus, some messages are used several times by different commands. Each function typically has messages of the following type:

■ for their usage

■ warning messages for "wrong" input or "inappropriate" usage

Most of the messages generated by *Mathematica* in "real calculations" relate to spelling warnings and potential "errors" in the input or in the computation.

Some words about "errors" are in order here. As a symbolic programming language, in *Mathematica* everything is an expression. (We discussed this point of view in detail in Chapter 2.) Expressions are characterized by their tree structure in a purely syntactic way. For many applications (but not all), it is not the syntactic but rather the semantic meaning that is of interest. The ability of *Mathematica* to return a closed form for `Integrate[`*func*`, `*var*`]`, the ability to calculate a larger determinant, and so on is often more important than it is to have a logarithm with five arguments, like `Log[1, 2, 3, 4, 5]`. At a syntactic level, the only thing that can go wrong is a sequence of characters that is not parsable. Here is an example of an unparsable expression.

```
1 @@ # ,, . :: ; " '' `` ~ ! 7 & '' // # * )) ]{}
```

The message generated in the last example indicates that *Mathematica* was not be able to construct an expression from the input. (In some sense this is the only "real error" that can happen. Any nonsysntax error could be considered a valid operation inside *Mathematica*. But for most purposes, a `$Failed` returned in case a file cannot be found will not be considered a successful operation.)

Once an expression has been parsed, it is an expression. Here is a syntactically correct input (although for most purposes it does not have much semantic meaning).

```
Sin[1, 2, 3] + 1[[-17]] + GCD[1.2, 9.6] - Cos["1"] Tan[Det[1, 2, 3]]/
        Function[1, 2] - Depth[] + 1[1]^2[I] + (1 < I)^Pi
```

The last input generated a couple of messages because the functions `Sin`, `Det`, and so on have a semantic meaning in *Mathematica*. As such, most functions expect a certain number of arguments of a specific type. It is largely a matter of opinion to call these messages "error" messages (in the same sense, it is an error to call `Sin` with more than one argument) or warning messages (from a syntactic point of view, everything is ok, but maybe the user intends to use the function `Sin` in a semantic way and not the expression `Sin[1, 2, 3]` in a purely syntactic way).

The phrasing of some of the last messages, like "is expected ", "must be", "invalid" are not to be taken too literally. Surely, a computer program does not have expectations and no legal action will be caused by defining a non-rule to be an option. These messages are hints for potential mistakes of users in the sense that these messages take for granted that functions that are doing mathematics are only called for this purpose and not as generic expressions to be manipulated in a structural way. A more technical (but for beginning users less helpful) phrasing of the messages would be "No built-in rule exists for the arguments …". (But this statement is trivially true for almost all arguments of almost all functions.)

Sometimes messages even make statements about nonexisting objects; they are phrased to direct the user to potential mistakes. The following input `"1` is not a *Mathematica* expression, and so surely does not have a head. But the message anyway speaks about a string that misses something, implicitly assuming the user's intention to enter a string.

```
"1
```

In general, it is not a good idea to use a built-in function with an "inappropriate" number or type of arguments. In addition to the annoying messages, one cannot be sure that later versions of *Mathematica* will behave the same way; extended versions of these functions might accept more and different arguments.

It is difficult to know—without knowing the intention of a piece of code—what exactly is an "error" in *Mathematica*. As said, a message typically "only" indicates that the "typical" use of a function with certain arguments is not possible. In most cases, the function returns unevaluated in such situations. Sometimes, the result will be `$Failed`. `$Failed` indicates that the intended operation did not work.

```
0 := 0
```

(Another example of an operation that returns `$Failed` is the attempt to open a nonexistent file.)

But at the same time, many instances exist in which one might expect *Mathematica* to give a message and *Mathematica* does not give one. The generic assumption about the type of a variable (any user created symbol) in *Mathematica* is that of a finite complex number (some functions make more specialized assumptions about the nature of their arguments). But nevertheless, inputs like `x + 5 I < y + 2 + 3 I` will not generate an error message (one could argue complex numbers cannot be compared). (The use of *a* < *b* the function `Less[`*a*`, `*b*`]` should be obvious; we will come back to this function in the next chapter.)

```
x + 5 I < y + 2 + 3 I
```

Similarly, the use of $\pi(i\,e)$ as an integration variable in the following definite integral will not produce messages, although one might argue that $\pi(i\,e)$ is not a "real" (or not "really" an) integration variable.

```
Integrate[1[2]^Pi[I E], {Pi[I E], 2, 4}]
```

Sometimes *Mathematica* functions are called with symbolic input and only later, the symbolic parameters are specified as numeric quantities. Some *Mathematica* commands issue messages in this case. Here is an attempt to generate a "symbolic" table. A message is generated.

```
Table[1, {n}, {n}]
```

After specifying a positive integer value for `n`, the last result evaluates just fine.

```
n = 2; %
```

Here is the scalar product between two "symbolic vectors". Although the two "symbolic vectors" are not actual vectors (they do not have the head `List`), no message is generated this time. (*a*.*b* is the shortform for `Dot[`*a*`, `*b*`]` and represents the scalar product of *a* and *b*.)

```
symbolicVector1.symbolicVector2
```

As a rule of thumb, messages are not generated for "symbolic" input if the function they appear in is used in classical mathematics. A scalar product is used in classical mathematics, so no message was produced in the last case. A table (a list) is not, so *Mathematica* produced a message.

Let us come back to the messages. We now check to see if a usage message is available for all system commands. The following program generates a list of all built-in commands not documented with an associated symbol `::usage`.

```
builtInFunctionsWithoutUsageMessage =
First /@ DeleteCases[{#, MessageName[#, "usage"]&[
       Unevaluated @@ ToHeldExpression[#]]}& /@
       (* the built-in commands *) systemCommands, {_, _String}];
```

Quite a few of these undocumented commands exist.

```
Length[builtInFunctionsWithoutUsageMessage]
```

Here is the first dozen.

```
Take[builtInFunctionsWithoutUsageMessage,  12]
```

And here is the last dozen.

```
Take[builtInFunctionsWithoutUsageMessage, -12]
```

The reader should, when possible, avoid using undocumented built-in functions (e.g., any of the functions from `builtInFunctionsWithoutUsageMessage`); or functions explicitly declared in their usage messages as internal functions in your programs; or functions explicitly declared in their usage messages as internal functions, because no guarantee exists that they will be included in later versions of *Mathematica*. Also, their behavior and syntax may change in the next version.

For the sake of compatibility, several *Mathematica* commands from earlier versions have been included in the current one. Using them generates a message saying that they are "obsolete". We now create a list of all messages involving the word obsolete (again, look at the result, not the programming).

```
((* turn off some messages *)
 Off[Part::partw]; Off[$$Media::obsym];
 Off[StringMatchQ::string]; Off[StringMatchQ::strs];)

(* find the obsolete symbols *)
Print[Cases[#[[1, 1, 1, 1]]& /@ Select[allMessages,
       StringMatchQ[#[[1, 2]], "*obsolet*"]&], _Symbol]];

((* turn on the above messages *)
 On[Part::partw]; On[$$Media::obsym];
 On[StringMatchQ::string]; On[StringMatchQ::strs];)
```

Messages generated more than three times in one evaluation are usually only printed three times if the message `General:stop` is enabled.

In the following example, the error message is printed three times, although the error occurs six times.

```
{Sin[x, y, 1], Sin[x, y, 2], Sin[x, y, 3],
 Sin[x, y, 4], Sin[x, y, 5], Sin[x, y, 6]}
```

Every particular message that would normally be generated more than three times because the corresponding problem happens more often is actually printed only three times while `General::stop` is on.

The `On` and `Off` commands can be used to "turn on" and "turn off" the printing of messages.

`On[`*symbol*`::`*message*`]`

allows the message *message* for the symbol *symbol* to be printed, provided it is generated during the computation of an expression.

`Off[`*symbol*`::`*message*`]`

prevents the printing of the message *message* for the symbol *symbol*, even if it is generated during the computation of an expression.

(The internal undocumented function `Internal`DeactivateMessages` allows to temporarily turn off all messages generates while evaluating the expression *expr* in `Internal`DeactivateMessages[`*expr*`]`.)

A spelling warning is generated if a symbol is introduced that is similar to an already existing symbol and the corresponding warning messages are on. (The messages `General::spell` and `General::spell1` have been turned off globally in the notebooks of the *GuideBooks* to avoid having many spelling warnings scattered through the notebooks.) These messages give warnings when a symbol is used for the first time, and this variable name is similar to the name of an already-used variable.

```
On[General::spell1]
```

```
aNewSymbol
```

```
aNewSimbol
```

Note that two spelling-related messages exist, `General::spell` and `General::spell1`. We now turn off this warning.

```
Off[General::spell1]
```

```
aNewSymbola; aNewSymbolb; aNewSymbolc; aNewSymbold;
aNewSymbole; aNewSymbolf; aNewSymbolg; aNewSymbolh;
```

If a turned-off message is evaluated again, it is enclosed in `$Off[]`. (Otherwise, it would return the string with the explicit message.) This result means that the current message is turned off.

```
General::spell1
```

The following example produces a message.

```
1[[2]]
```

But `Part::partd` does not return the message content "`Part specification … is longer than depth of object`".

```
Part::partd
```

The reason that `Part::partd` did not evaluate to the corresponding string is that this special message is not a message of `Part`.

```
Messages[Part]
```

It is a message associated with `General`.

```
General::partd
```

User-defined messages can, in complete analogy to built-in messages, be created using `MessageName`. Here is a simple example for the user-defined function `myMsg`.

```
myMsg::toymess = "Now printing a MMeessaaggeee";
```

Here is the `FullForm` of the last expression.

```
Hold[myMsg::toymess = "Now printing a MMeessaaggeee"] // FullForm
```

Messages associated with `General` are typically used by many functions, and to avoid repetition, they are present only once.

A user-defined message can be printed at the appropriate time using `Message`.

> `Message[`*symbol*`::`*name*`]`
>
> prints the message *name* associated with the symbol *symbol*.

```
a = 1; b = 2; Message[myMsg::toymess]; a b
```

We currently have no definition made for `myMsg`.

```
??myMsg
```

One message is currently associated with `myMsg` through `Messages`.

```
Messages[myMsg]
```

In connection with our earlier discussion, we still need to explain the meanings of `HoldPattern` in the last result. It has appeared several times in connection with upvalues and downvalues.

> `HoldPattern[`*expression*`]`
>
> is equivalent to *expression* as a pattern, but does not evaluate *expression*.

No expressions inside `HoldPattern` are evaluated, because of the `HoldAll` attribute of `HoldPattern` attributes.

```
Attributes[HoldPattern]
```

`HoldPattern` is necessary here to create the correspondence between the name of a message and the message, because "the result" of the calculation of *symbol*`::`*message* is just the contents of the message, as in the following second example.

```
HoldPattern[myMsg::"toymess"]
```

```
myMsg::"toymess"
```

The function `HoldPattern` is used by internal (and user-defined) functions to prevent evaluation while still allowing pattern matching. We see that `HoldPattern` is necessary if we look at the result of the above constructions with `HoldPattern` dropped from `HoldPattern[myMsg::"toymess"]`. The left-hand side evaluates to the right-hand side `myMsg::"toymess"` disappeared. We come back to `HoldPattern` in the next chapter when we discuss patterns in detail.

Next, we look at the meaning of the semicolons in `; … ; … ;`. We encountered such structures already repeatedly, so it is time to discuss them. We cannot get at the `FullForm` of "`;`" directly.

```
FullForm[a; b; c]
```

But here is the result with `Unevaluated`.

```
FullForm[Unevaluated[a; b; c]]
```

Any function with a `Hold`-like attribute makes it possible to see the head `CompoundExpression`.

```
FullForm[Hold[a; b; c]]
```

> `CompoundExpression[`$expression_1$`, `$expression_2$`, …, `$expression_n$`]`
>
> or

expression$_1$`;` *expression*$_2$`;` `…;` *expression*$_n$

represents one compound expression whose individual components are *expression*$_1$, *expression*$_2$, …, *expression*$_n$. All the n expressions will be evaluated, but only the result of *expression*$_n$ will be returned. Side effect outputs (like carrying out `Print` statements and displaying graphics) will be generated.

Note the difference between `a; b` and `a; b;`. The latter is understood as `a; b; Null`.

```
{FullForm[Hold[a; b]], FullForm[Hold[a; b; ]]}
```

Although nothing is returned by `Null`, the line number of the *Mathematica* inputs nevertheless increases in the following inputs.

```
Null
```

```
Null
```

A `Null` is always inserted between two commas. (Because it is relatively seldom that we want `Null` as an argument, *Mathematica* gives a warning message here.)

```
functionWithThreeNullArguments[ , , ] // FullForm
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

■ 4.1.2 A Program that Reports on Functions

Let us go on and discuss how to get information on more than one command at one time. To do this we use attributes, as discussed in the last chapter. The command `Attributes` also carries the attribute `Listable`.

```
Attributes[Attributes]
```

Here are the current attributes of the functions `Information`, `Messages`, and `Options`.

```
Attributes[Information]
```

```
Attributes[Messages]
```

```
Attributes[Options]
```

We now add `Listable` to the attributes of these three commands.

```
SetAttributes[Information, Listable];
SetAttributes[Messages   , Listable];
SetAttributes[Options    , Listable];
```

This input makes them listable; that is, they automatically apply to lists of elements.

```
Attributes[{Information, Messages, Options}]
```

We now introduce an expression `nameList[Fi]`, which evaluates the list of all names beginning with `Fi`.

```
nameList[Fi] = Names["Fi*"];
Length[nameList[Fi]]
```

Next, we define a command `allAttributes` that finds all of the attributes of the elements in its argument, which should be a list.

```
allAttributes[list_] := Attributes[Evaluate[ToExpression[list]]]
```

Before we use this function, we briefly elaborate on its implementation. Here, we have linked `Evaluate` and `ToEx`⋱ `pression`, which ensures that we get the attributes for *list*, and not those of `ToExpression[`*list*`]`, because `Attri`⋱

butes has the attribute HoldAll. We have used ToExpression because Names gives a String and not an expression, as we can see in the following example.

```
Names["Plo*"][[1]]

Head[%]

FullForm[%%]
```

The head String was mentioned already in Chapter 2; we now discuss its relation to expressions in more detail. Strings, like numbers, are fundamental objects. It is not possible to assign any values to them.

```
"iam11" = 11
```

ToExpression["*expression*"]
converts the String "*expression*" into the symbol *expression*, which can be manipulated.

Here, we convert "1 + 2 + 3" into the *Mathematica* expression 1 + 2 + 3, which is then evaluated as 6.

```
ToExpression["1 + 2 + 3"]

Head[%]
```

The following input returns #1^2, not 4. The reason is that at the time the pure function substitutes 2 for its dummy variable, no explicit Slot[1] is present. The Slot[1] appears at this time only inside a string and not as a *Mathematica* expression. Then the pure function gets evaluated, meaning the string "#1^2" gets converted to the expression #1^2.

```
ToExpression["#1^2"]&[2]
```

Often, we want to prevent the immediate computation of a string that has been converted to a *Mathematica* expression. This action is possible with ToHeldExpression.

ToHeldExpression["*expression*"]
converts the String "*expression*" into the expression *expression* without doing any further evaluation, and resulting in Hold[*expression*].

In the following, the expression 1 + 2 + 3 is not evaluated.

```
ToHeldExpression["3 + 2 + 1"]
```

When "*expression*" is not a syntactically correct expression, $Failed is returned.

```
ToHeldExpression["+ 1 +"]
```

Another, and in general more flexible and powerful, way to convert a string to an unevaluated expression is the following command.

ToExpression["*expression*", *form*, *function*]
converts the string "*expression*" into the expression *expression* by using the interpretation of the format type *form*. The function *function* is applied to the resulting expression before any further evaluation.

Here, the three-argument version of ToExpression converts the string "3 + 2 + 1" into an unevaluated expression.

```
ToExpression["3 + 2 + 1", InputForm, Hold]
```

Any other function with a Hold-like attribute will result in an unevaluated expression.

```
ToExpression["3 + 2 + 1", InputForm, Unevaluated]
```

A Sequence disappears inside Hold.

```
ToExpression["Sequence[1, 2, 3]", InputForm, Hold]
```

Inside HoldComplete, a Sequence can survive.

```
ToExpression["Sequence[1, 2, 3]", InputForm, HoldComplete]
```

Be aware that the expression is neither computed nor reordered into the canonical normal form. But ToHeldExpres⋱ sion does not convert every expression in the form "*expression*" into Hold[*expression*]. In view of the way in which the HoldAll attribute of Hold works, as we have discussed in Chapter 3, evaluation happens in the following example.

```
ToHeldExpression["Evaluate[1 + 1]"]
```

Hold often affects the appearance of an expression somewhat. With the command HoldForm, we can make the enclosing "holder" invisible.

```
ToExpression["1 + 1", InputForm, HoldForm]
```

```
FullForm[%]
```

The reverse, that is, the conversion of a *Mathematica* expression into a String, is accomplished by ToString.

ToString[*expression*]
converts the Symbol *expression* into the String *expression*.

Note that in the conversion of a *Mathematica* expression into a String, it is best to start with the InputForm; the formatted OutputForm frequently does not give what we want.

```
testExpression = Integrate[x^2 Exp[-4/5x^2], x]
```

If testExpression is formatted by *Mathematica*, it appears in the usual way.

```
ToString[testExpression]
```

In the FullForm, we see, however, that it contains \n for new lines and that the expression is enclosed in quotes.

```
FullForm[%]
```

The same statement holds for the TreeForm.

```
FullForm[ToString[TreeForm[testExpression]]]
```

StandardForm uses an efficient box notation.

```
FullForm[ToString[StandardForm[testExpression]]]
```

TraditionalForm also uses an efficient box notation.

```
FullForm[ToString[TraditionalForm[testExpression]]]
```

The following form is often more appropriate for most applications. It is short, readable, and one-dimensional (1D), and it uses only ASCII characters.

```
FullForm[ToString[InputForm[testExpression]]]
```

We can, of course, also produce a string of the full form of testExpression.

```
ToString[FullForm[testExpression]]
```

After the last side steps, we now discuss the function `allAttributes` defined above. Here is what it does.

```
allAttributes[list_] := Attributes[Evaluate[ToExpression[list]]]

allAttributes[nameList[Fi]]
```

This input shows that `ToExpression` and `Evaluate` are both necessary.

```
Attributes[nameList[Fi]]

Attributes[ToExpression[nameList[Fi]]]
```

It might happen that a command evaluates something other than itself. (See the examples below.) We discuss how to treat this case appropriately in Chapter 6.

We can now get all options in an analogous way.

```
Options[Evaluate[ToExpression[nameList[Fi]]]]
```

Because `Information` does not give an `Out[`*i*`]`, we indeed get all information (i.e., a short description of the command, its attributes, and its options), but we cannot immediately operate further on this text with *Mathematica*. (To save space we use only the first six commands from `nameList[Fi]`.)

```
Information[Evaluate[ToExpression[nameList[Fi][[{1, 2, 3, 4, 5, 6}]]]]];
```

Moreover, the formatting leaves something to be desired; at least, some blank lines should be between the different commands. (We come back to this in Chapter 6 after our discussion of the ways in which lists can be manipulated.) Meanwhile, the reader can use the following code fragment. (Warning: this produces a huge output.)

```
(* read relevant files *)
Get[ToFileName[{$TopDirectory, "SystemFiles", "Kernel", "TextResources",
                $Language}, #]]& /@ {"Messages.m", "Usage.m"};

(* allow to extract all information *)
ClearAttributes[#, {Protected, ReadProtected}]& /@
             ((Unevaluated @@ #)& /@ (ToHeldExpression /@ Names["*`*"]));

Information /@ Names["*`*"]
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

4.2 Control over Running Calculations and Resources

■ 4.2.1 Intermezzo on Iterators

In this subsection, we present the `Do` command for iterative calculations and discuss the general iterator notation of *Mathematica*.

`Do[`*loopBody*`, ` *iterator*$_1$`, ` *iterator*$_2$`, ..., ` *iterator*$_n$`]`

repeats the calculation of the expression *loopBody* as often as described by *iterator*$_1$, *iterator*$_2$, … , *iterator*$_n$. The order of the iteration is from right to left, which means the rightmost iterator is the innermost one.

Iterators work from left to right, which means the leftmost iterator variable is localized first, then the second leftmost is localized, then the third leftmost, and so on. The current value of the leftmost iterator can influence the limits of the other iterators. Here is a first simple example.

```
Do[Print["Now printing ", i, " and ", j], {i, 3}, {j, 2}]
```

In the next example, the starting value of the inner iterator is 12.

```
Do[Print["Now printing ", i, " and ", j], {i, 3}, {j, 12, 12 + i}]
```

The next input uses the same iterator variable for the inner and the outer loops. The inner one overwrites the value of the outer one.

```
Do[Print[{i, i}], {i, 2}, {i, 3}];
```

The next input uses again the same iterator variables for the inner and the outer loops. In addition, the upper limit of the inner loop is depending on the value of outer loop variable. (This use of iterator variables is confusing and should be avoided.)

```
Do[Print[{i, i}], {i, 2}, {i, i}];
```

The following constructions can serve as iterators:

`{`n_{max}`}`	repeats n_{max} times
`{`n`, `n_{max}`}`	n runs from 1 to n_{max} in steps of size 1
`{`n`, `n_{min}`, `n_{max}`}`	n runs from n_{min} to n_{max} in steps of size 1
`{`n`, `n_{min}`, `n_{max}`, `n_{step}`}`	n runs from n_{min} to n_{max} in steps of size n_{step}

Because `Do` does not result in a printed or a returned expression (it actually returns `Null`, which is not given as output), we still need `Print` to see what actually happens.

```
Do[j = i, {i, 2, 5}]

FullForm[%]
```

In the next input, the argument of `Print` is computed (i.e., the expression `j = i` is evaluated) for every value of `i`.

```
Do[Print[j = i], {i, 2, 5}]
```

The current value of `j` is 5.

```
j

Do[Print[j = i], {i, -2, -5, -1}]
```

Now, the current value of `j` is -5.

```
j
```

Here, there is nothing to do, because $-2 > -5$ and we cannot step from -2 to -5 in steps of size +1.

```
Do[Print[j = i], {i, -2, -9, 1}]
```

`Null` was the result, which is always suppressed in the output.

```
FullForm[%]
```

The number of steps to be carried out is calculated as $\lfloor (n_{max} - n_{min}) / n_{step} \rfloor$ before the first loop is started and, at this point, must be equal to a positive integer. Thus, for example, the following constructions are all possible.

```
Do[Print[j], {j, i - 1, i + 1}]
```

```
Do[Print[j], {j, -E, Pi}]
```

```
Do[Print[j], {j, 0.3, 1.2, 0.456789}]
```

The evaluation of $\lfloor (n_{max} - n_{min}) / n_{step} \rfloor$ will be carried out purely numerically, and in a purely numerical calculation, it is not possible to decide if ⌊6 - 2 Sqrt[2] - (Sqrt[2] - 1)^2⌋ is equal to 2 or 3. In such cases, a warning message is issued.

```
Do[Print[j], {j, (Sqrt[2] - 1)^2, (2 - 2 Sqrt[2] + 1) + 3}]
```

The “correct” number of iterations is carried out in the last example. By changing the last input slightly, we can get the wrong number of iterations (but, again, *Mathematica* gives a warning about a potentially wrong number of steps).

```
Do[Print[j], (* use 3 - 2 Sqrt[2] written in different forms *)
   {j, (Sqrt[2] - 1)^2, (2 - 2 Sqrt[2] + 1) + 3 - 10^-500}]
```

In the next case, just one value will be assumed for `j`.

```
Do[Print[j], {j, 0, 0}]
```

In our next example, `I` $= i = \sqrt{-1}$, but the difference between the upper and lower limits is a positive real number > 1 (head `Integer`). So, it is an allowed iterator construction.

```
Do[Print[j], {j, I, I + 3}]
```

Here, the imaginary part cancels completely.

```
Do[Print[j], {j, 1.0 + I, 3.0 + I}]
```

A tiny imaginary part is ignored.

```
Do[Print[j], {j, 1.0 + 1.0 I, 3.0 + 1.0 I, 1/2}]
```

```
Do[Print[j], {j, 1 + (N[1, 20] + 10^-20) I, 3 + 1 I, 1/2}]
```

The following construction leads to an error message. At first glance, the difference between the upper and lower limits appears to be a positive number, but *Mathematica* evaluates the upper limit to be `Infinity`, before the difference is computed (but returns the original input).

```
Do[Print[j], {j, Infinity, Infinity + 3}]
```

Here, the difference between the upper and lower limits is not a positive integer.

```
Do[Print[j], {j, 2, 4 I}]
```

In all of these cases, the input is returned unchanged, even if *Mathematica* has done some intermediate computations.

```
FullForm[%]
```

The reason that *Mathematica* can return the original expression, including its unevaluated arguments, is the attribute `HoldAll` of `Do`, which allows `Do` to keep a copy of its original, unevaluated arguments.

```
Attributes[Do]
```

Note the behavior of `Do` when the step size is explicitly 0.

```
Do[Print[i], {i, 1, 1, 0}]
```

```
Do[Print[i], {i, 0, 0, 0}]
```

Here is a comparison with step size 1.

```
Do[Print[i], {i, 0, 0}]
```

```
Do[Print[i], {i, 1, 1}]
```

The iterator in `Do` is computed at the beginning, and then the first argument of `Do` is operated on. Later (meaning carried out at runtime in the body of `Do`) changes have no effect on the iterator.

Here is an unsuccessful attempt to alter the step size during the computation. The number of iterations and the values of the iterator variables are calculated before the iterations are actually carried out. Because iterators use a `Block`-like scoping (see below), it is nevertheless possible to change the value of the iterator variable inside the loop for each iteration.

```
j = 1; Do[j = 5i; i = i - 1; Print["i = ", i, ", j = ", j], {i, 0, 5, j}]
```

We make sure that the first argument of `Do` is assigned a concrete value of the running (dummy) variable `j`.

```
j
```

Also, built-in symbols can be used as iterator variables (but this is not a good programming style and so, we will not make much use of this possibility). In the following input, we use `Pi` as the iterator variable.

```
Do[Print[Pi^2], {Pi, 2, 4}]
```

Everything we have stated about the behavior of `Do` for various forms of the iterators also holds for similar commands with the same iterator notation (e.g., `Sum`, `Product`, and `Table`).

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

■ 4.2.2 Control over Running Calculations and Resources

After this brief detour involving `Do` and iterators, we now come back to the main theme of this subsection. A calculation can be stopped interactively with `Quit`, which kills the *Mathematica* kernel, or more smoothly with `Abort`.

`Abort[]`

stops the running calculation "as soon as possible" after `Abort[]` appears.

In the following example, `C` will not be printed.

```
Do[Print[A]; Print[B]; Abort[]; Print[C]; {4}]
```

`Abort[]` can be overridden with `AbortProtect`.

`AbortProtect[`*expression*`]`

prevents the aborting of the computation of *expression* if `Abort[]` is encountered in computing *expression*.

Now `C` is printed out, but the result of the entire calculation is still `$Aborted`.

```
AbortProtect[Print[A]; Print[B]; Abort[];
                     (* restore state here *) Print[C]]
```

A common use of `AbortProtect` is inside a user-defined function, in which system functions (like `$RecursionLimit`, `$IterationLimit`, `$MaxExtraPrecision`) are set to nonstandard values or large expressions are generated. In such cases, we do not want these values of system functions globally visible after aborting a calculation.

Such restoring of original values of system variables and removing temporary variables is also the reason that, under some circumstances, aborting (using `Abort[]`) might take a substantial time.

`CheckAbort` can be used to check to see if an abort will be encountered.

`CheckAbort[`*expression*`,` *anAbortOccurred*`]`

gives the result of the computation of *expression* if no abort was encountered; otherwise, it gives *anAbortOccurred*. If an `Abort` command is encountered, the computation stops at that point.

This result is what we get for the above example.

```
CheckAbort[Print[A]; Print[B]; Abort[]; Print[C],
           (* restore a proper Mathematica state here *)
           Print["An abort has occurred!"]]
```

To interrupt a calculation and to continue at another point, we can use the pair of functions `Throw` and `Catch`.

`Throw[`*expression*`,` *throwTag*`]`

sends the expression *expression* to the nearest enclosing `Catch` whose second argument matches *throwTag*. In case *throwTag* is omitted, the nearest enclosing `Catch` receives *expres*⁚ *sion*.

`Catch[`*expression*`,` *catchTag*`]`

returns the first argument of the `Throw` inside *expression* whose tag matches *catchTag*. If *catchTag* is omitted, the first argument of any executed `Throw` in *expression* is returned.

Here is a simple example that uses the one-argument forms of `Throw` and `Catch`. `Throw[a]` returns the current value of `a` to the outer `Catch`. The assignment `a = 3` is never executed.

```
Catch[a = 1; a = 2; Throw[a]; a = 3]
```

In the next example, again the `Throw` after the assignment `a = 2` is executed. But the outer `Catch` has the tag `ais3` which does not match the throw tag `ais2`. As a result the whole `Throw` is returned in `Hold`.

```
Catch[a = 1; a = 2; Throw[a, ais2]; a = 3;
                    Throw[a, ais3]; a = 5,
      ais3]
```

To allow `Catch` to compare the thrown tag with its second argument, it must have the attribute `HoldFirst`. This allows to second argument to be evaluated before the first.

```
Attributes[Catch]
```

Another pair of functions that similarly to `Throw` and `Catch` cooperate in a nested manner is the pair of functions `Sow` and `Reap`.

`Sow[`*expression*`,` *sowTag*`]`

indicates the expression *expression* to be collected by the next enclosing `Reap` whose second argument matches sow*Tag*. If *sowTag* is omitted, the nearest enclosing `Reap` will collect *expression*.

`Reap[`*expression*`, ` *reapTag*`]`

returns a list of the value of *expression* and the first arguments of all occurrences of `Sow` inside *expression* whose tags match *sowTag*. If *reapTag* is omitted, the first arguments of any `Sow` evaluated in *expression* are returned.

Here is again a simple example. The result of `Reap[...]` is the a list of two elements. The first element is the value 3 which comes from the last `Sow[a]` and the second element contains the three values of `a` that were sown while evaluating the first argument of `Reap`.

```
Reap[a = 1; Sow[a]; a = 2; Sow[a]; a = 3; Sow[a]]
```

In the next example, we use `Sow` with a second argument. The outer `Reap` is used without a second argument. As a result, the returned second element is a list whose elements are the sown expressions for each sow tag.

```
Reap[a = 1; Sow[a, 1]; a = 2; Sow[a, 2]; a = 3; Sow[a, 3]]
```

If we only want to reap the sown expressions for a special tag, we use a second argument in `Reap`.

```
Reap[a = 1; Sow[a, 1]; a = 2; Sow[a, 2]; a = 3; Sow[a, 3], 1]
```

Often, we want to limit the time and memory resources to be used in the computation of an expression. (Some built-in functions do this, for instance, `Simplify`.)

`TimeConstrained[`*expression*`, ` *seconds*`]`

stops the computation of *expression* after *seconds* seconds, provided it is still running.

`MemoryConstrained[`*expression*`, ` *bytes*`]`

stops the computation of *expression* if more than *bytes* bytes are used.

The abort will not always happen exactly after the prescribed amount of time, or if the prescribed amount of memory is exceeded, but "as soon after as possible". Here, we abort two extensive, although elementary, calculations. $3\,333\,333^{333\,333}$ cannot be calculated by using only 100 bytes.

```
MemoryConstrained[333333^333333, 100]
```

Already, the result needs nearly 1 MB storage space.

```
ByteCount[3333333^333333]
```

If the reader does not have a quantum computer, the next calculation should abort.

```
TimeConstrained[
        Nest[Integrate[#, x]&,
             Sin[x^12 + Exp[x + Sqrt[x]]] Tan[x], 12345], 1]
```

Frequently, we want to know whether messages have been generated during a computation.

`Check[`*expression*`, ` *messageOccurred*`]`

gives the result of the computation of *expression*, if during its computation no message was generated; otherwise, it gives *messageOccurred*.

In the following example, a message is generated.

```
Check[Do[i = j, {j, 1, 2, 3, 4, two}],
      Print["Was there a typo in the iterator?"]]
```

Here, everything works fine.

```
Check[Print[Do[i = j, {j, 1, 4, 2}]],
      Print["There was no typo in the iterator."]]
```

The following command provides an overview of the resources used in a *Mathematica* session.

`MemoryInUse[]`

gives the current amount of memory in bytes currently used by the *Mathematica* kernel.

`MaxMemoryUsed[]`

gives the maximum amount of memory in bytes used by the *Mathematica* kernel in a session.

`TimeUsed[]`

gives the total CPU time in seconds used by the *Mathematica* kernel for calculations (not including PostScript interpreter times or times used by other subprocesses).

So far, we have used the following amounts of memory and CPU time.

```
MemoryInUse[]

MaxMemoryUsed[]

TimeUsed[]
```

To reduce the amount of memory currently needed to store all expressions, we use `Share`.

`Share[]`

usually reduces the amount of memory needed and returns the size of freed memory.

`Share` works as follows: All symbols in the symbol table are checked, and those with the same values are coupled with cross references. An automatic function similar to `Share[]` is built into the *Mathematica* kernel, although it is not always called. Thus, it is sometimes a good idea to call `Share` manually from time to time. In this connection, the following command is of interest.

`ByteCount[`*expression*`]`

gives the number of bytes of memory required to store *expression*.

For example, to store the antiderivative of $x^{66}\cos(x)^{66}$ requires around 1 MB.

```
ByteCount[cosInt1 = Integrate[x^66 Cos[x]^66, x]]
```

Here is its size measured in *Mathematica* subexpressions.

```
LeafCount[cosInt2 = Integrate[x^66 Cos[x]^66, x]]
```

Now, we have two expressions that have exactly the same value, `cosInt1` and `cosInt2`. If we now run `Share`, we can considerably reduce the memory currently used.

```
MemoryInUse[]

Share[]

MemoryInUse[]
```

The small difference between the value returned by `Share[]` and the explicit difference is caused by the state changes of the second `MemoryInUse[]` call.)

```
%%% - %
```

Σ (* session summary *) **`TMGBs`PrintSessionSummary[]`**

4.3 The $-Commands

■ 4.3.1 System-Related Commands

The following commands give information on the version of *Mathematica* being used.

`$VersionNumber`

gives the version number of the *Mathematica* implementation.

```
$VersionNumber
```

In programs, we sometimes use constructions like
`If[$VersionNumber <= 5.1,` *doSomethingThatCanNotBeDoneInEarlierVersions*`,`*giveAMessage*`]`.

`$Version`

gives the version of the *Mathematica* kernel being used.

```
$Version
```

`$CreationDate`

gives the date when the version of *Mathematica* being used was created in the form of a list `{`*year*, *month*, *day*, *hour*, *minute*, *second*`}`.

```
$CreationDate
```

The output of the date is in the typical form.

`Date[]`

gives the current date in the form of a list `{`*year*, *month*, *day*, *hour*, *minute*, *second*`}`.

```
Date[]
```

Mathematica has a software-implemented high-precision arithmetic. Whenever possible, it will use machine arithmetic. Various properties of the machine arithmetic can be inferred from the following commands.

`$MachineEpsilon`

gives number of the type `Real` that, when added to the machine real number 1.0, gives a result larger than 1.0.

For the computer in use here, this input shows the number.

```
$MachineEpsilon
```

Here is a test of the defining property of `$MachineEpsilon`.

```
a1 = (1.0 + $MachineEpsilon);
a2 = (1.0 + $MachineEpsilon/2);
{a1 - 1.0, a2 - 1.0}
```

Here is the number of digits used in working with machine accuracy.

`$MachinePrecision`

gives the number of digits to be carried in a calculation with machine numbers.

```
$MachinePrecision
```

The use of `N[`*expr*`, $MachinePrecision + 1]` will result in carrying out a numerical evaluation of *expr* using *Mathematica*'s high-precision arithmetic. In distinction to a machine number, for a high-precision number all digits are explicitly displayed.

```
N[Sqrt[2], $MachinePrecision]
```

```
N[Sqrt[2], $MachinePrecision + 1]
```

An related command to `$MachineEpsilon` produces the largest machine number.

`$MaxMachineNumber`

gives the largest number that can be used with machine arithmetic.

Here is this number.

```
$MaxMachineNumber
```

Multiplying the last number by 2 results in a high-precision number. (High-precision numbers are used by *Mathematica* if either a number has more digits or is larger in size so that it cannot be represented as a machine number.) And a high-precision number displays all its digits.

```
2 %
```

Dividing the last result by `2` yields a number identical to the original one in size, but now it is a high-precision number (all its significant digits are displayed).

```
%/2
```

Mathematica also computes with larger numbers, but not directly via hardware arithmetic. Here is an example.

```
1111111111111111111^121 2^222222 1.189731495357231766 10^4932
```

The use of high-precision arithmetic results in a loss of speed. Here are the computational times required for the computation of $\left(2.0 \times 10^{10\, exp}\right)^{0.2}$ as a function of the exponents. We clearly see that the average growth of the time has a (first) big increase at the exponent *exp* of around the switching to high-precision arithmetic.

```
Log[10, $MaxMachineNumber]/10
```

The reader should look primarily at the result, and not at the program. (The thin vertical line marks the exponent *exp* whose computation leads to a number greater than `$MachinePrecision`.)

```
ListPlot[Table[{exp, (* the timing *)
                Timing[Do[(2.0 10^(10 exp))^0.2, {1000}]][[1, 1]]/10},
               {exp, 0, 300}],
         AxesLabel -> {"exp", "time/seconds"}, AxesOrigin -> {0, 0},
         PlotRange -> All, PlotStyle -> {PointSize[0.01]},
         (* vertical line at the largest machine number *)
         GridLines -> {{Log[10, $MaxMachineNumber]/10}, None}]
```

The smallest machine number can be obtained with `$MinMachineNumber`.

`$MinMachineNumber`

gives the smallest number that can be used with machine arithmetic.

Here is the current value.

```
$MinMachineNumber
```

Squaring the last number again creates a high-precision number.

```
%^2
```

Taking the square root of the last number yields again a high-precision number.

```
Sqrt[%]
```

In addition to the largest machine real number, sometimes we need to know the largest machine integer. The function `$MaxMachineInteger` from the context `` Developer` `` is returning this number. (We will discuss the meaning of a context in Subsection 4.6.5.)

```
Developer`$MaxMachineInteger
```

```
Log[2, %] // N[#, 22]&
```

Most *Mathematica* iterators require the number of steps to be a machine integer. So the following `Do` loop generates a message.

```
sum = 0;
Do[sum = sum + k,
   Evaluate[{k, 10^100,
             10^100 + Developer`$MaxMachineInteger + 1}]]
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

■ 4.3.2 Session-Related Commands

We have already seen at least one of the commands to be treated in this subsection (as a result of `Abort`).

`$Aborted`

is the result of breaking off a computation either with `Abort[]` or interactively.

`$Line` is another session-related function.

`$Line`

gives the number of the current input line.

`$Line` can also be set by the user, but then all information contained in the previously used inputs and outputs is no longer available. Symbols, function definitions, and so on, remain in effect, however.

```
?In
```

We get all definitions attached to `In` (which means all previous inputs) with `DownValues` (or with `?In`).

```
DownValues[In]
```

So, we can use `In` like any other function definition. Like many other built-in commands, `In` has attributes.

```
Attributes[In]
```

Calling an already-stored `In[`*n*`]` results in the reevaluation of the corresponding input.

```
In[3]
```

To start the line numbering again from 1, we input the following.

```
$Line = 1
```

```
2 + 2
```

Be aware that when resetting the line number, we do not change the state of the whole *Mathematica*, so variables declared before this change are still available.

In this connection, note the difference in the display of `In[`*number*`] :=` and `Out[`*number*`]=`. The inputs are associated with `In` via `SetDelayed`, whereas the outputs are associated with `Out` via `Set`. Thus, by inputting `In[`*number*`]`, we reevaluate the input (which may have been evaluated earlier) corresponding to the current values of the parameters or global variables. (Also `Out[`*number*`]` reevaluates of course.)

```
a = 1; b = 2;
```

```
a + b
```

```
b = 3
```

```
In[$Line - 2]
```

Both inputs via `In` and outputs via `Out` are stored as `RuleDelayed`-objects (to be discussed in the next chapter) in the `DownValues` of `In`, respectively, `Out`.

```
DownValues[In] // First
```

```
DownValues[Out] // First
```

So `In` is a symbol like any other one in *Mathematica*. As such, we can define values for certain arguments. Here we set the value of `In[1111]` to be the current input line number.

```
Unprotect[In];
```

```
In[1111] := $Line
```

```
In[1111]
```

```
In[1111]
```

A `$`-command that is important not for `In`, but for `Out` is `$HistoryLength`.

`$HistoryLength`

gives the number of last outputs that should be stored with `Out`.

Currently, all outputs are stored in the `DownValues` of `Out`.

```
$HistoryLength
```

We can reset the value to, say, 2.

```
$HistoryLength = 2
```

Now, only the last two outputs can be retrieved and the `%%%` stays unevaluated.

```
{%, %%, %%%}
```

The use of a small `$HistoryLength` (typically 0) value is especially recommended in case of large outputs, like graphics. The actual graphic is a “side effect”, and `Out` still contains the *Mathematica* description of the graphics. We will reset the value of `$HistoryLength` a few times in the Graphics volume [27✱]. For now, we reset `$History⋱Length` to its default value.

```
$HistoryLength = Infinity
```

To collect the messages generated by inputs, we have `$MessageList`.

`$MessageList`

gives a list of the messages originating during the evaluation of the current input.

`$MessageList` gives the messages in a form allowing them to be further manipulated.

Here are some simple functions with the “wrong” number of arguments.

```
meLi = (Sin[1, 2, Log[1, 2, 3, 4], Log[5, 6, 7, 8], 4]; $MessageList)
```

The names of the messages are included in `Hold`.

```
FullForm[%]
```

For operations connected with graphics, the following command is important.

`$DisplayFunction`

gives the system information needed to draw an image (where and how to plot it).

Here, its current value is shown. (For details about the command `$DisplayFunction`, see Chapter 1 of the Graphics volume [27✱].)

```
$DisplayFunction
```

Next, we discuss two `$` commands that are important in connection with recurrence and iteration: `$Recursion⋱Limit` and `$IterationLimit`. Here is a simple recurrence formula to compute a function `caf`.

```
caf[1] = 1;
caf[n_] := caf[n - 1] n^2 - 2n
```

Unfortunately, although it is algorithmically completely correct, the formula produces error messages when we try to calculate `caf[298]`.

```
caf[298];
```

Here is the reason for these error messages.

`$RecursionLimit`

gives the maximum number of recurrence steps to be carried out for recursive function definitions. `$RecursionLimit` can be set to `Infinity`, which allows an arbitrary number of iterations.

The default value of `$RecursionLimit` is 256.

```
$RecursionLimit
```

If we make `$RecursionLimit` sufficiently big, we can compute `caf[298]` without an error message.

```
$RecursionLimit = 500
```

```
caf[298]
```

On the other hand, the following still does not work.

```
caf[550];
```

If the reader is not working on a Unix-running computer, he should exercise some care in dealing with very recursive calculations, because they make heavy use of the stack. Such calculations can easily crash *Mathematica*. Here is an example (we do not run it here, of course) involving the so-called Ackermann function ([1✱], [11✱], [2✱], [23✱], [30✱], [17✱], [19✱], [21✱], [20✱], [25✱], [22✱], [8✱], [9✱], [24✱], [18✱], and [10✱]).

```
f[a_, 0] = 0;
f[a_, 1] = 1;
f[a_, i_] := i;
g[a_, b_, 0] = a + b;
g[a_, 0, i_] := f[a, i - 1];
g[a_, b_, i_] := g[a, g[a, b - 1, i], i - 1]

(* calculate an example *)
$RecursionLimit = Infinity;
g[a, 2, 2]
```

We now move from recursion to iteration. The following flawed function definition leads to an overstepping of the iteration limit.

```
f[x_] = f[x]
```

The number of iterations can be limited using `$IterationLimit`.

`$IterationLimit`

gives the number of iteration steps to be carried out in iterative computations. (`$IterationLimit` can be set to `Infinity`, which allows an arbitrary number of iterations.)

In the above example, increasing `$IterationLimit` does not help; this function definition simply goes on forever. We do not go into a discussion about the difference between iteration and recursion here, but will come back to it soon.

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

4.4 Communication and Interaction with the Outside

■ 4.4.1 Writing to Files

The exchange of data and commands between *Mathematica* and other programs is accomplished using the *MathLink* standards (which we do not treat here; see [32✱]). InterCall (http://analytica.com.au/Products/InterCall.html) can communicate with external Fortran and has been designed to interface with numeric libraries, such as NAG and IMSL.

In this subsection, we will discuss how to save definitions on a file and how to load them in again. `Definition` and `FullDefinition` are useful *Mathematica* commands for working with other programs.

`Definition[`*symbol*$_1$`,` *symbol*$_2$`, …,` *symbol*$_n$`]`
gives the definition of the user-defined symbols *symbol*$_1$, *symbol*$_2$,…, *symbol*$_n$ (more precisely, all such symbols that do not carry the attribute `ReadProtected`).

`FullDefinition[`*symbol*$_1$`,` *symbol*$_2$`, …,` *symbol*$_n$`]`
gives the complete recursive definition of the user-defined symbols *symbol*$_1$, *symbol*$_2$,…, *symbol*$_n$ along with all other symbols contained in them (more precisely, all such symbols that do not carry the attribute `ReadProtected` or `Protected`).

Here is a little example showing the difference between `Definition` and `FullDefinition`. Here, `f` is defined via `g`, `g` via `h`, and `h` is defined recursively.

```
f[x_] := g[x]^2;
g[y_] := h[y]^2;
h[z_] := h[z] = h[z - 2] + h[z - 1];
h[0] = 0;
h[1] = 1;
```

Now, we calculate `f[4]`.

```
f[4]
```

Here is the immediate definition of `f`.

```
Definition[f]
```

Here is the complete definition of `f` (including the special values for `h`).

```
FullDefinition[f]
```

For later reuse, in *Mathematica* or in another program, the `InputForm` is more useful (a mimic of the “conventional” mathematical notation is not employed, only ASCII characters are used).

```
InputForm[FullDefinition[f]]
```

Giving `g` the attribute `Protected` avoids the definitions for `g` from being given.

```
SetAttributes[g, Protected]
```

```
FullDefinition[f]
```

`Definition` also produces a result for the two arguments `In` and `Out`. Indeed, unless `$Line` has been explicitly manipulated, we get both the inputs and the outputs for the current session. Here is an example.

```
Definition[In]

Definition[Out]

FullForm[%]
```

The last output contains `Definition[f]` and `FullDefinition[f]`. `Definition` and `FullDefinition` do not return the function definition(s) explicitly via `Out`, but instead act as a formatting device. Here is a definition for *f*.

```
f[x_] := g[x]^2;
g[x_] := 4;

FullDefinition[f]
```

The fullform has the head `FullDefinition`.

```
FullForm[%]
```

And the depth of the last expression is just 2.

```
Depth[%]
```

Similar to functions like `TreeForm` that only act as a formatting device, `FullDefinition[f]` also is a formatting device. When used as an argument in other functions it allows us to obtain the inputform as a string.

```
InputForm[FullDefinition[f]] // ToString // InputForm
```

Changing the definition of *g* and reevaluating the above output gives prints the current definition of *g*.

```
g[x_] := 6;
%%%%%
```

Because `Out` contains all of the results obtained up to the current time in a given *Mathematica* session, the amount of stored information can be huge, especially if a (large) number of plots have been created. This space can be freed using these commands.

```
Unprotect[Out]; Clear[Out]; Protect[Out];

Unprotect[Out]; DownValues[Out] = {}; Protect[Out];
```

But, of course, the associated information is lost. `%`, `%%`, `Out[n]`, will no longer work as before. To avoid building a large list if outputs, we can set the value of `$HistoryLength` to a small value.

To write to external files, we can use `Put`.

`Put[`*expression*$_1$`,` *expression*$_2$`, ...,` *expression*$_n$`, "`*fileName*`"]`
or, if $n = 1$,

expression `>>` `"`*fileName*`"`
writes *expression*$_1$, *expression*$_2$, … , *expression*$_n$ to the file *fileName*.

Here is a test.

```
Put[InputForm[FullDefinition[f]],
    "PutTestFileWithAUniqueFileNameHopefully"]
```

To read files, we use `Get`.

`Get["`*fileName*`"] or << "`*fileName*`"`
reads the file *fileName*. This form is also used to read *Mathematica* packages.

To see whether this function works, we erase the symbols for `f`, `g`, and `h` along with their values.

```
Unprotect[g];
Remove[f, g, h];
Print[FullDefinition[f]]
```

Now, we read the definitions back in.

```
<< "PutTestFileWithAUniqueFileNameHopefully";
FullDefinition[f]
```

To delete files from within *Mathematica*, we have `DeleteFile`.

`DeleteFile["`*fileName*`"]`
deletes the file *fileName*.

Mathematica also has the functions `RenameFile` and `DeleteDirectory` to rename files and to delete directories. In addition, the functions `CopyFile`, `CopyDirectory` to copy files and directories exist. We will occasionally make use of these functions.

We now delete the file `PutTestFileWithAUniqueFileNameHopefully`.

```
DeleteFile["PutTestFileWithAUniqueFileNameHopefully"]
```

If we want to write out some temporary files in the default directory for temporary files of the computer system, we can use the function `OpenTemporary`. Here we write a trigonometric identity (as a string) to a temporary file.

```
tempFileStream = OpenTemporary[]

WriteString[tempFileStream,
  "Cos[Pi/17] == Sqrt[(15 + Sqrt[17] + Sqrt[34 - 2*Sqrt[17]] +
   Sqrt[2*(34 + 6*Sqrt[17] - Sqrt[34 - 2*Sqrt[17]] +
   Sqrt[34*(17 - Sqrt[17])] - 8*Sqrt[2*(17 + Sqrt[17])])])/2]/4"]

Close[tempFileStream]
```

Reading the identity back into the kernel gives a `$MaxExtraPrecision::meprec` message (see Chapter 5) because the identity cannot numerically disproved.

```
Get[tempFileStream[[1]]]
```

If we try to read a nonexistent file (e.g., with `<<"PutTestFileWithAUniqueFileNameHopefully")` after the above deletion, we get an error message of the form `Get::noopen: Can't open PutTest`.

The effect of `Put[InputForm[FullDefinition[`*expression*`]], "`*fileName*`"]` can also be obtained in the following shorter way.

`Save[`*expression*$_1$`, `*expression*$_2$`, …, `*expression*$_n$`, "`*fileName*`"]`
appends
`InputForm[FullDefinition[`*expression*$_1$`, `*expression*$_2$`, …, `*expression*$_n$`], "`*fileName*`"]`
to the file *fileName*.

`Put` overwrites existing files. To append to existing files, we can use `PutAppend`.

`PutAppend[`*expression*$_1$`, ` *expression*$_2$`, …, ` *expression*$_n$`, "`*fileName*`"]`

or, if $n = 1$,

expression `>>> "`*fileName*`"`

adds *expression*$_1$, *expression*$_2$, …, *expression*$_n$ at the end of the file *fileName*.

Σ (* session summary *) **`TMGBs`PrintSessionSummary[]`**

■ 4.4.2 Simple String Manipulations

The following string operations are often very useful, especially in connection with other programs because they allow us to create arbitrary formatted input for these other programs. String operations are also very useful inside *Mathematica* (for instance, for file name manipulations and creation of special symbol names) and for the program-based creation of variable names within *Mathematica*.

`StringJoin["`*string*$_1$`", "`*string*$_2$`", …, "`*string*$_n$`"]`

or

`"`*string*$_1$`" <> "`*string*$_2$`" <> … <> "`*string*$_n$`"`

combines the strings "*string*$_1$", …, "*string*$_n$" into a single string.

`StringLength["`*string*`"]`

gives the number of string characters in *string*.

`StringReplace["`*string*`", {"`*stringOld*$_1$`" -> "`*stringNew*$_1$`",`
`"`*stringOld*$_2$`" -> "`*stringNew*$_2$`", …,`
`"`*stringOld*$_n$`" -> "`*stringNew*$_n$`"}]`

replaces the substrings "*stringOld*$_i$" in the string "*string*" by "*stringNew*$_i$". For only one replacement, the outside pair of braces can be dropped.

`StringTake["`*string*`", {`*n*`}]`

gives the first *n* characters of "*string*".

`StringReverse["`*string*`"]`

reverses the order of the characters in "*string*".

`StringPosition["`*string*`", {"`*subString*`"}]`

gives the position of the substring in "*string*". The result is a list containing lists of the beginning and end locations of the desired "*subString*".

Two of these `String` commands also have options.

```
Options[StringReplace]

Options[StringPosition]
```

Here, we discuss only one of these options, namely, `IgnoreCase`.

`IgnoreCase`

is an option for several string manipulation functions.

Default:

`False` (differentiate between lowercase and uppercase letters)

Admissible:

`True` (uppercase and lowercase letters are treated the same)

Here is a little example involving a string manipulation command. First, we input six strings.

```
s1 = " Once";
s2 = " there";
s3 = " was";
s4 = " a";
s5 = " Mathematica";
s6 = " session, in which ...";
```

Then, we join them into one string.

```
StringJoin[s1, s2, s3, s4, s5, s6]
```

Here, the constructed string is backward.

```
StringReverse[%]
```

This string consists of 51 individual characters.

```
StringLength[%]
```

Next, we find the places where an `"e"` appears.

```
StringPosition[%%%, "e"]
```

Here are the places where an `"er"` appears.

```
StringPosition[%%%%, "er"]
```

Next, we replace all `a`'s by `e`'s, and vice versa. (The meaning of `"e" -> "a"` should be obvious; we treat the `Full`⋱`Form` of `->` in the next chapter.)

```
StringReplace[%%%%%, {"m" -> "o", "e" -> "a"}, IgnoreCase -> True]
```

Here, just the lowercase letters are replaced.

```
StringReplace[%%%%%%, {"m" -> "o", "e" -> "a"}, IgnoreCase -> False]
```

An important application of `String` operations is to format data and/or commands to be passed back and forth between *Mathematica* and other programs (e.g., line length, first position in a line, etc.).

To end this subsection, we give an example of what can be done with the `ToString` command. We create a short program that does nothing other than print itself—a “classical” problem for any computer language.

```
Print[ToString[#0][]] & []
```

How does it work? The pure function `Print[ToString[#0][]]` is called with zero arguments. Then, the pure function is evaluated. `#0` represents the function itself, so the argument of `Print[ToString[#0][]]`, `ToString[#0][]` is printed. The result of this evaluation is `Null`, because it is a `Print` statement. `ToString` comes into play in a twofold way. First, the quotes are not printed in `StandardForm`, so a `String` printed looks the same as the corresponding symbol. Second, without `ToString`, the result of the evaluation of the pure function

`Print[#0[]]` would be `Print[#0[]]`, which is itself, and again, this would be evaluated and so on, which means we would have an infinite recursion. Using `InputForm`, we see the quotes from the string.

```
Print[InputForm[ToString[#0][]]] & []
```

An obvious generalization would be a program that prints itself more than once, for instance, two times.

```
Do[Print[ToString[#0][]], {2}] & []
```

We discussed strings, but which characters are allowed in a string? In addition to the ASCII characters, *Mathematica* supports many more characters, like Greek letters and many special mathematical symbols. Their `InputForm` looks like the character.

```
{InputForm[α], InputForm[ℝ], InputForm[…]}
```

Their `FullForm` shows their names. Character names have the form `\[`*name*`]`.

```
FullForm[%]
```

The result of the following calculation returns all named characters in *Mathematica*.

```
allNamedCharacters =
  Drop[Select[FromCharacterCode /@ Range[10^5],
              Characters[ToString[FullForm[#]]][[-2]] === "]"&], 2];
```

Mathematica has more than 1100 special characters.

```
Length[allNamedCharacters]
```

Here are the first 50. (Not all may display on every computer. To see all of them, the corresponding fonts must be installed.)

```
{#, FullForm[#]}& /@  Take[allNamedCharacters, 50]
```

Here are the last 50.

```
{#, FullForm[#]}& /@  Take[allNamedCharacters, -50]
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

■ 4.4.3 Importing and Exporting Data and Graphics

Mathematica can import and export data and graphics from and to a variety of file formats. Here is a list of the currently supported formats for import and export.

```
$ImportFormats
```

```
$ExportFormats
```

The actual import and export of files is carried out using the functions `Import` and `Export`.

`Import["`*fileName*`", "`*format*`"]`

imports the file *fileName* and returns the corresponding *Mathematica* expression.

`Export["`*fileName*`", `*toBeExportedExpression*`, "`*format*`"]`

exports the expression *toBeExportedExpression* to the file *fileName* using the file format *format*.

Because the *GuideBooks* do not come with GIFs, we use the GIFs that come with *Mathematica*. The following input find and imports all files with the extension gif from the *Mathematica* installation directory.

```
importedGifsThatComeWithMathematica =
    Import /@ FileNames["*.gif", $InstallationDirectory, Infinity];

Length[importedGifsThatComeWithMathematica]
```

We display them.

```
(* group graphics into o in one row and fill last row *)
groupGraphicsAndShow[l_] := Show[GraphicsArray[#]]& /@
With[{o = 3},
 Module[{λ = Length[l], μ = Mod[Length[l], o], P = Partition[l, o]},
       Which[μ == 0, P,
             λ <= o, l,
             True, Append[P, Join[Take[l, -μ], Table[{}, {o - μ}]]]]]]

groupGraphicsAndShow @ importedGifsThatComeWithMathematica
```

Here the same is done with files with the extension .JPG.

```
importedJpgsThatComeWithMathematica =
    Import /@ FileNames["*.jpg", $InstallationDirectory, Infinity]

groupGraphicsAndShow @ importedJpgsThatComeWithMathematica
```

Next, we import a webpage. The page to be imported contains todays papers deposited at the *Arxiv* preprint server in quantum physics.

```
newInQuantumPhysics = Import["http://arxiv.org/list/quant-ph/new", "Text"];

Short[newInQuantumPhysics, 12]
```

The next input extracts the titles of the papers and prints them.

```
CellPrint[Cell["∘ " <> StringReplace[#, "\n "->" "], "PrintText"]]&/@
      (StringTake[#, {11, -4}]& /@ StringCases[newInQuantumPhysics,
                              ShortestMatch["Title:</B>" ~~ __ ~~ "<BR"]]);
```

Sometimes one wants to carry out conversions completely within *Mathematica*, without reading from or writing to files. The two functions `ImportString` and `ExportString` come in handy here.

`ImportString["`*string*`", "`*format*`"]`
imports the string *string* and returns the corresponding *Mathematica* expression.

`ExportString[` *toBeExportedExpression*`, "`*format*`"]`
exports the expression *toBeExportedExpression* to a string using the file format *format*.

Here is a simple parametrized surface.

```
pp3d = ParametricPlot3D[{Sin[x], Cos[y], Sin[2 x + y]},
                        {x, 0, 2Pi}, {y, 0, 2Pi},
                        Boxed -> False, Axes -> False]
```

We generate the string corresponding to this graphics in EPS format.

```
pp3dEPS = ExportString[pp3d, "EPS"];
```

Here are the first few lines of the resulting string.

```
Short[pp3dEPS, 8]
```

Importing the string yields a *Mathematica* `Graphics` expression. (Be aware that we started with a genuine 3D graphics of head `Graphics3D`, but now have a 2D graphics expression of head `Graphics`.)

```
ImportString[pp3dEPS, "EPS"]
```

Visually the imported graphic looks identical to the original one.

```
Show[%]
```

The conversion details from and to the various file formats are regulated through the option `ConversionOptions`. For a detailed listing of the possible suboption settings, see the help browser pages for `Import` and `Export` `Import` and `Export`. The next example exports the above 3D graphic is a low-quality JPEG. The file is quite small, about 6 kB.

```
pp3dJPEG = ExportString[pp3d, "JPEG",
                        ConversionOptions -> {"Quality" -> 2}];
ByteCount[pp3dJPEG]
```

Importing and displaying the graphic shows now clear differences to the original graphic.

```
Show[ImportString[pp3dJPEG, "JPEG"]]
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

4.5 Debugging

Because programming errors are bound to occur in writing longer programs, it is important to have a way to find them [33✱]. In *Mathematica*, the currently two most important “tools” for debugging are `On` and `Trace` (in addition to sprinkling `Print` statements throughout the code to be debugged).

`On[]`

shows every step in the computation of an expression explicitly (except for “very internal” ones). This output is not in a form that can be immediately processed, because it is not attached to the form `Out[… ]`, but instead appears “between the lines”.

`Off[]`

cancels the effect of `On[]`.

Here is what this looks like for the computation of $\xi = \frac{\pi}{4}$ followed by $\sin(\pi + (2 + 3)\,\pi + \xi)$.

```
On[];
ξ = Pi/4;
Sin[Pi + (2 + 3) Pi + ξ]
Off[]
```

Note the `-->` arrow inside the individual steps of the calculation. This arrow is not a *Mathematica* command.

Debugging with `On[]` can lead to exceptionally large printed output. Moreover, the (wall clock) runtime increases dramatically.

We can use `On[]` to give a detailed look at what is going on with respect to variable renaming when calculating `Function[x, Function[y, x^2 y][2]][3]`. In the first step, `3` is substituted for `x` inside the inner function, and the `y` of the inner function is renamed (to make sure that it does not interfere with any other variable). The resulting expression 3^2 inside the inner function is not evaluated (HoldAll is an attribute of Function). In the second step, 2 is substituted for y$ (we come back to this renaming issue soon again) and the resulting expression 2×3^2 is evaluated.

```
On[]

Function[x, Function[y, x^2 y][2]][3]

Off[]
```

In the next example, the occurrences of the two semicolons ; results in the evaluation of a CompoundExpression.

```
On[]; one = 1; two = 2;

Off[]
```

No Out[] appears in either line, because of Null. We have already encountered Null, but not yet discussed it.

Null

is a symbol returned by functions that work by side effects (e.g., Print, Do, etc.), or as a filler in certain expressions that take multiple subexpressions, when a subexpression was not given (e.g., argument lists and in CompoundExpression).

When Null is the final result of a computation, it is not displayed as an output—this enables one to suppress output one wants to hide (e.g., by creating a CompoundExpression with an implicit trailing Null by applying a semicolon to your input). Here are some examples of the appearance of Null in the output.

```
Print[3; ]

myFunction[a, b, , d, e]
```

Here, we see where "Null prevents itself from being printed as output": No associated Out-result is visible.

```
Null

%

FullForm[%]
```

Often, Trace is much more appropriate than is On[].

Trace[*expression*]

gives a list (with head List) of all intermediate results in the computation of *expression*. The result of Trace is output via Out[…] as a nested list and can be further manipulated and analyzed with *Mathematica*.

Although Trace returns a result that can be further manipulated (in contrast to the printing generated by On[…]), it may be very deeply nested (we quickly get to several hundred levels of braces of the form {*firstEvaluated*{*second*-*Evaluated*{*thirdEvaluated*{*fourthEvaluated*{…}}}}}). But the list returned by Trace is a syntactically correct *Mathematica* expression and can be analyzed by *Mathematica*. This "machine analysis" is particularly useful in larger calculations.

```
Trace[ξ = Pi/4; Sin[Pi + (2 + 3) Pi + ξ]]
```

The individual subexpressions are enclosed in HoldForm to prevent their further evaluation and do not possess a visible Hold.

```
FullForm[%]
```

The computation of the integral $\int^x x^2 \sin^4(x)\cos^3(x)\ln(x)\,dx$ involves a lot of intermediate steps.

```
tr = Trace[int = Integrate[x^2 Sin[x]^4 Cos[x]^3 Log[x], x]];
```

We do not look at the complete `Trace` result.

```
Short[tr, 5]
```

Instead, we analyze its structure.

```
{Depth[tr], ByteCount[tr], LeafCount[int],
 StringLength[ToString[FullForm[tr]]]}
```

Here is the same analysis done for a definite integral.

```
tr = Trace[
 int = Integrate[x^2 Sin[x]^4 Cos[x]^3 Log[x], {x, 0, Pi}]]
```

We do not look at the complete `Trace` result.

```
Short[tr, 5]
```

Instead, we analyze its structure.

```
{Depth[tr], Length[tr], ByteCount[tr], LeafCount[int],
 StringLength[ToString[FullForm[tr]]]}
```

`Trace` also has options.

`Trace` has nine options.

```
TraceAbove     TraceBackward   TraceDepth
TraceForward   TraceInternal   TraceOff
TraceOn        TraceOriginal   MatchLocalNames
```

Along with the following, these `Trace` options greatly simplify the debugging problem.

```
TraceAction   TraceDialog   TraceLevel
TracePrint    TraceScan
```

We do not consider all options of `Trace` and related commands here, but instead look only at two examples using some of the `Trace` commands.

`TracePrint[`*expression*`]` prints all expressions originating from the computation of *expression*.

```
TracePrint[3 + t + 5 + 2 5]
```

By setting the option `TraceInternal -> True`, we usually get as detailed a protocol as with the use of `On[]`. (Integration may be mentioned as an exception, for instance, `On[]; Integrate[Exp[x^3], x]` leads to a much longer result than does `Trace[Integrate[Exp[x^3], x]]`). Here is the result of `Trace`.

```
Trace[Integrate[Exp[x^3], x]]
```

Now, we use `On[]` to follow the calculation. To avoid getting a large amount of printouts, we temporarily suppress the printouts and collect the single steps in the list `bag`. (How the following program works will be discussed in Chapter 6.)

```
(* keep where messages are sent to *)
old$Messages = $Messages;
(* a bag for collecting the steps *)
bag = {};
(* as a side effect, collect all steps *)
$MessagePrePrint = AppendTo[bag, #]&;
(* redirect messages *)
$Messages = nowhere;
On[];
(* do the integration *)
Integrate[Exp[x^3], x];
Off[];
(* restore where messages are sent to *)
$Messages = old$Messages;
$MessagePrePrint = Short;
```

Inside `bag`, we collected a lot of information about the more than 6000 steps that were carried out.

```
{Depth[bag], Length[bag], ByteCount[bag], LeafCount[bag],
 StringLength[ToString[FullForm[bag]]]}
```

Here are the last recorded steps of the evaluation of $\int e^{x^3}\,dx$ that used `On[]`.

```
Take[bag, -12]
```

With `Trace`, it is easy to see the difference between iteration and recursion. Recursion determines the depth (as measured by `Depth`) of results of `Trace`; iteration determines their length (as measured by `Length`). We begin with a recursive definition. (We will reset `$RecursionLimit` and `$IterationLimit` to prevent large printouts. We save the current value for later use.) These are the current values for `$RecursionLimit` and `$IterationLimit`.

```
oldValues = {$RecursionLimit, $IterationLimit}
```

We now change them temporarily and do a very recursive and a very iterative calculation. Using `Trace`, we can monitor how the calculation performs. Then, we look at the length and depth of the list generated by `Trace`.

```
Clear[f];
$RecursionLimit = 100;
$IterationLimit = 200;
f[1] = 0;
f[n_] := f[n - 1] + n;
recursiveTrace = Trace[f[50]];
{Depth[recursiveTrace], Length[recursiveTrace]}
```

Next, we give a failed iterative definition.

```
Clear[g];
$RecursionLimit = 200;
$IterationLimit = 100;
i = 1;
g[n_] := g[n];
iterativeTrace = Trace[g[50]];
{Depth[iterativeTrace], Length[iterativeTrace]}
```

Now, the depth is greater than the length of this list. Here is an iterative calculation with `FixedPoint`.

```
tr1 = Trace[FixedPoint[(100 # + 1/#)/101&, 10.]];
{Depth[tr1], Length[tr1]}
```

`Nest` is another function carrying out a purely iterative calculation.

```
Function[tr, {Depth[tr], Length[tr]}][
                   Trace[NestList[Sin, 1``12, 1000]]]
```

We reset `$RecursionLimit` and `$IterationLimit` to their old values.

```
{$RecursionLimit, $IterationLimit} = oldValues
```

A general computation contains both recursive and iterative elements.

`Input`, `InputString`, and `Interrupt` are also often very useful for interactive debugging.

`Input[]`

reads in a *Mathematica* expression interactively.

`InputString[]`

reads in a `String` interactively.

`Interrupt[]`

stops the program and displays a menu of choices to proceed interactively.

Because all three commands are partially machine dependent (and require further interactive input), we do not illustrate them here.

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

4.6 Localization of Variable Names

■ 4.6.1 Localization of Variables in Iterator Constructions

`Sum` and `Product` are two other typical constructions, in addition to `Do`, that involve iterator variables. Their syntax is nearly self-explanatory.

`Sum[`*term*`,` *iterator*`]`

forms the sum of the summation terms *term* corresponding to the running variables in *iterator*. Here, *iterator* is used in the usual iterator notation.

`Product[`*term*`,` *iterator*`]`

forms the product of the factors *term* corresponding to the running variables in *iterator*. Here, *iterator* is used in the usual iterator notation.

We now define a function for computing the sum of the first n powers x^i ($i = 1, \ldots, n$). (The following is the classical example in *Mathematica* in which the iterator variables and the independent variables will coincide.)

```
PowerSum[x_, n_] := Sum[x^i, {i, n}]
```

Here it works as expected.

```
PowerSum[x, 7]
```

Here it does not.

```
Clear[i];
PowerSum[i, 3]
```

The last result is largely caused by the behavior of the function `SetDelayed`. As discussed in the last chapter, an instance of a pattern variable will be substituted in the right-hand side, which leads to the expression `Sum[i^i, {i, 3}]` that evaluates to 32.

```
Sum[i^i, {i, 3}]
```

Often, $term_i$ is defined first outside of `Sum[`$term_i$`, `*iterator*`]` in the form $term_i = something(i)$ and then "inserted" in `Sum`, `Do`, or `Product`. This behavior is exemplified below.

```
term = k^3 + k^2 + k + 1;
Sum[term, {k, 1, 4}]
```

The result 144 can be easily understood if we look at the following sum.

```
(1^3 + 1^2 + 1 + 1) + (2^3 + 2^2 + 2 + 1) +
(3^3 + 3^2 + 3 + 1) + (4^3 + 4^2 + 4 + 1)
```

Here is an example concerning the order of localization of the iteration variables and the assignment of their limits. At every stage (where the first stage in the following is in the `Table`, then in `Sum`, and last in `Product`), the iteration variable is localized, and then the upper limit is computed.

```
i = 3;
Table[Sum[Product[i^i, {i, i}],
          {i, i}],
      {i, i}]
```

Here is the same result in a somewhat more understandable iterator notation.

```
l = 3;
Table[Sum[Product[i^i, {i, j}],
          {j, k}],
      {k, l}]
```

Here is the detailed calculation for comparison.

```
{1^1, 1^1 + 1^1 2^2, 1^1 + 1^1 2^2 + 1^1 2^2 3^3}
```

As we shall see in the following subsections, variables can also be protected in other ways.

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

■ 4.6.2 Localization of Variables in Subprograms

Often, it is convenient to use the same variable names in subprograms as in the main program without worrying that variables interfere with each other in some way. This scoping can be accomplished in *Mathematica* using `Block`, `Module`, and `With`.

`Block[{`x_1`, `x_2`, `…`, `x_n`}, `*program*`]`

or

`Block[{`$x_1 = x0_1$`, `$x_2 = x0_2$`, `…`, `$x_n = x0_n$`}, `*program*`]`

creates a local environment in which to run the program *program*. Before the call of `Block`, values assigned to the symbols x_i are temporarily erased (if necessary, the values $x0_i$ are assigned). After the computations in `Block` are finished, the x_i are reset to their old values.

`Module[{`x_1`, `x_2`, `…`, `x_n`}, `*program*`]`

or

`Module[{`$x_1 = x0_1$`, `$x_2 = x0_2$`, `…`, `$x_n = x0_n$`}, `*program*`]`

creates a local environment in which to run the program *program*. When the module is called, the symbols x_i are temporarily replaced by new variables with the internal form x_i*$uniqueNumber*. If necessary, they are initialized to the values $x0_i$. After completion of the commands in the module, these variables are removed, unless they have been exported to the outside.

`With[{`$x_1 = x0_1$`, `$x_2 = x0_2$`, `…`, `$x_n = x0_n$`}, `*program*`]`

creates a local environment in which to run the program *program*. When `With` is called, all instances of the symbols x_i in *program* are replaced by the local constants $x0_i$. The x_i cannot be assigned any further new values inside *program*.

`Block` is a dynamic scoping construct. This means the values of variables are local to `Block`. Here is a simple example. The Print statement inside Block prints ψ because no valued was assigned to ψ inside `Block`.

```
ψ = 1;
Block[{ψ}, Print[ψ]];
ψ
```

The next input uses `Block` to define the highly recursive function [26✱]

$$V_r(x) = \frac{1}{2}\,V_{r-1}(x)^2 + \frac{V_{r-1}\left(x^2\right)}{1 - \sum_{k=0}^{r-1} V_k(x)}$$

$$V_0(x) = x.$$

Each time the function `𝒱[`*n*`, `*x*`]` is called, the local definitions for `V` are evaluated. The definitions contain a `Set`⋱`Delayed[Set[...]]` construction to cache intermediate values. Then `V[`*n*`, `*x*`]` is evaluated and returned. After leaving the `Block`, all definitions made for `V` are no longer existent.

```
𝒱[n_Integer, x_] :=
Block[{V},
      V[0, z_] := z;
      V[r_, z_] := V[r, z] =
      1/2 (V[r - 1, z]^2 + V[r - 1, z^2])/
                        (1 - Sum[V[k, z], {k, 0, r - 1}]);
      (* calculate value with actual n and V *)
      V[n, x]]
```

Calculating `V[10, `*x*`]` yields 55 cached values for `V`. The next input calculates `𝒱[10, 2]` using machine-arithmetic, high-precision arithmetic, and exact arithmetic. Due to the complicated iterative nature of the rational function `𝒱[10, `*x*`]`, the machine-precision result suffers from cancellation errors.

```
{𝒱[10, 2.], 𝒱[10, N[2, 100]], 𝒱[10, 2] // N}
```

No definition for the earlier cached values of `V` exists anymore.

```
?V
```

Avoiding too many cached values is sometimes of importance for memory reasons. Without the above `Block[{...}, `*localDefinitionsWithCaching*`]`, the following graphic displaying the phase of `𝒱[6, `*z*`]` over the complex *z*-plane would accumulate more than three million cached values.

```
ContourPlot[Arg[𝒱[6, x + I y]]^2/Pi^2, {y, -2, 2}, {x, -2, 2},
       PlotPoints -> 400, ColorFunction -> (Hue[0.8 #]&),
       PlotRange -> All, Contours -> 20, Compiled -> False,
       ColorFunctionScaling -> False, ContourLines -> False]
```

In the definitions above, *program* is either a single expression or an expression with head `CompoundExpression`. We now demonstrate the use of `Module` by looking again at the `PowerSum` example discussed above. In the following construction, the iterator variable is localized, preventing any interference with other variables.

```
ModulePowerSum[x_, n_] := Module[{i}, Sum[x^i, {i, n}]]
```

The summation now works even with `i` as the index. (Note the result for the third call, in which the upper limit on the exponents continues to be called `n`, but the local summation variable is renamed.)

```
{ModulePowerSum[x, 5], ModulePowerSum[i, 5], ModulePowerSum[n, n]}
```

If we had also used `i` on the left-hand side, no assignment would have been possible for nonsymbols inside `Sum`.

```
ModulePowerSumWithi[x_, i_, n_] := Module[{i}, Sum[x^i, {i, n}]]
```

Here, the iterator variable is a symbol.

```
ModulePowerSumWithi[x, i, 4]
```

In the next two cases, the iterator variable is not a symbol and the creation of a local variable inside `Module` fails.

```
Clear[i, j, x];
ModulePowerSumWithi[x, j[2], 4]
```

```
ModulePowerSumWithi[x, 3, 4]
```

The following input does not give the “intended” result, because the summation variable `k` is localized to `k$`*integer* and has nothing to do any longer with the `k` from `term`.

```
term = k^3 + k^2 + k + 1;
Module[{k}, Sum[term, {k, 1, 10}]]
```

An amusing example of constructing an exceptionally long name can be added based on a) `Nest` and b) the property of `Module` to create new variable names.

```
Nest[Module[{#}, #]&, x, 100]
```

By looking at their attributes, we can verify that the variables created in `Module` are only temporary in existence.

```
Module[{x}, Print[Attributes[x]]];
```

Because no `x` was explicitly exported from the `Module`, `x` now has no attributes.

```
Attributes[x]
```

`Temporary`

is an attribute to identify variables created inside of `Module` and other scoping constructs. This attribute results in the removal of these variables when they are no longer needed (i.e., when the computations in the `Module` are complete), provided they have not been explicitly exported.

In the following example, we use the variable `temp` inside of `Module`. Inside the `Module`, we print a list of all names matching `"temp*"`.

```
Module[{temp}, Print[Names["temp*"]]; temp = 2^2]
```

After completion of `Module`, the temporary version of `temp` has vanished (the variable `temp` was created when parsing the whole `Module`).

```
Print[Names["temp*"]]
```

The attribute `Temporary` does not cause variables of the form *name$number* to be removed if they have been exported, this is shown in the following example.

```
Remove["x*", z]

Module[{x1, x2}, z = {x1 + x2}]

Names["x*"]
```

Now, we remove the variable `z`.

```
Remove[z]

??z
```

This process did not remove the variables `x1`, `x2` and their local copies from `Module`.

```
Names["x*"]
```

They still carry the attribute `Temporary`. (They carry the attribute independent of their environment.)

```
Function[argument, Attributes[argument], {Listable}][%]
```

When we also clear the content of `Out`, they no longer exists.

```
Unprotect[Out]; Clear[Out]; Protect[Out];
Names["x*"]
```

The attribute `Temporary` works only for variables inside `Module`. And inside `Module`, the attribute is automatically given. So the following attempt to use a variable `x` with attribute `Temporary` inside `Block` fails.

```
Remove[x]

Block[{x}, SetAttributes[x, Temporary]; x; 1]

??x
```

While the first arguments of `Block`, `Module`, and `With` contain syntactically `Set` or `SetDelayed` statements, because of the variable localization to be achieved, no real assignments as discussed in the last chapter are carried out. The following input demonstrates this by temporarily disabling `Set`. Although `Set` is disabled, the local variable `a` has the value 1.

```
Function[scoper, Block[{Set}, Print @ scoper[{a = 1}, b = a]],
         Listable] @ {Block, Module, With};
```

The variables *listOfVariables* appearing in the first argument of `Function[`*listOfVariables*`,` *function*`]` are also local. Concerning renaming of variables, we recall a remark from Chapter 3.

> `Function` uses a construction in some sense similar to `Module` internally to protect its "dummy" variables.

Here is a function definition for `𝕋` that is nested.

```
𝕋 = Function[y, Function[x, x^2 + y^2]]
```

Two arguments can be given to the function `𝕋.` We get a function if we give only one argument (this function can then get a further argument). Then, the remaining one carries the typical `$` inside of `Function`.

```
𝕋[x]
```

If the variables inside `Function` end with a `$`, things can go wrong.

```
𝕋 = Function[y, Function[x$, x$^2 + y^2]]
```

Now, the dummy variable `x$` of the inner `Function` is no longer properly renamed.

```
𝕋[x$]
```

Now, the dummy variable `x$` of the inner `Function` is no longer different from the supplied argument.

```
𝕋 = Function[y, Function[$, $^2 + y^2]]
```

```
𝕋[$]
```

For pure functions that use `#` no renaming can happen.

```
Function[y, Function[#^2 + y^2]][#]
```

The last example shows that user symbols should never end with `$`. An analogous construction for manually creating "new" variables exists.

`Unique[{`x_1`,` x_2`, …,` x_n`}]`
creates a list of new variables of the form `{`x_1\$*number*`,` x_2\$*number*`, …,` x_n\$*number*`}`, so that no overlap exists with already existing variables. With only one variable, the braces `{ }` are not needed.

Here three new variables are formed from the "old" `newVar`, `x`, and `y`.

```
Unique[{newVar, x, y}]
```

We now give two simple examples of the use of `With` (we come back to its use in the next subsection). Here is one typical application of `With`. `With` constructs "local constants".

```
Clear[a, b, x, y];
x = 1;
```

```
With[{x = 9, y = (a + b)^9 // Expand}, (y - x)^x]
```

All symbols appearing in the first argument (head `Symbol`) are localized. Essentially, it does not matter how the variables are named.

```
With[{Hold = 33, Exit = 44, Quit = 55, I = 66, NotebookOpen = 77},
     Hold Exit Quit[] I Symbol NotebookOpen[]]
```

Note that `Goto` commands also belong to the subject of subprograms and program structure. *Mathematica* includes `Goto` and `Label`, as well as `Catch` and `Throw`. If possible, the use of the two commands `Goto` and `Label` should be avoided, because code containing `Goto` is typically is difficult to read. We have not used them in any of the examples implemented in this book, and so, we do not bother to discuss them here. If the reader decides that he needs to use them, remember that their behavior is different from that in other programming languages. The reader should make sure to read the documentation carefully.

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

■ 4.6.3 Comparison of Scoping Constructs

We present a detailed comparison of the various possibilities for creating subprograms using `Module`, `Block`, and `With` in this subsection. This comparison is very important for practical applications.

`Block` initializes only the values of the variables, not the variables themselves. `Module` initializes the variables themselves by creating new variables of the form *var*\$. `With` introduces local constants and replaces all literal occurrences of the variables in its body.

To illustrate this difference, we give the variable `testVar` the value 1111.

```
testVar = 1111
```

In the following `Block`, we make `testVar` a local variable; the result of `Block` is 2222, and afterward the variable again has the same value as beforehand.

```
Block[{testVar},
      testVar = 2222;
      Print["The current value of testvar inside Block is: ", testVar];
      testVar]
```

```
testVar
```

With no value assignment, we get the value 1111.

```
Block[{testVar}, testVar]
```

With `Print`, we can see that `testVar` is assigned the value 1111 only after the `Block` has been completed (the line `Block::trace: Block[{testVar}, Print[testVar]; testVar] --> testVar` from tracing is relevant here.)

```
Block[{testVar}, Print[testVar]; testVar]
```

Using `On[]` also shows that inside `Block` `testVar` has no value.

```
On[];
Block[{testVar}, Print[testVar]; testVar]
Off[];
```

The following input shows that the variable name in `Block` remains unchanged, and no `$` is appended.

```
Block[{testVar}, Hold[testVar]]
```

For comparison, we now perform the same operations with `Module`.

```
Module[{testVar}, testVar = 2222; Print[testVar]; testVar]
```

```
testVar
```

```
Module[{testVar}, Print[testVar]; testVar]
```

```
Module[{testVar}, Hold[testVar]]
```

This example indicates that every call of `Module` results in the creation of a new variable *var*`$`*number*. Even if the variables are not explicitly exported, the number in *var*`$`*number* is incremented.

```
Do[Module[{a}, Print[ToString[a]];], {5}]
```

In order to avoid double use (one from the user and one from `Module`) of variable names, the user should not introduce variables with names of the form *var*`$`*number*.

The three scoping constructs `Block`, `Module`, and `With` allow also for delayed assignments in their first arguments. This is especially relevant if the result of the right-hand side of the assignment can change. Here is a simple example.

```
Block[{𝕕 := Date[]}, {𝕕, Pause[5]; 𝕕}]
```

When using `Module` with initializations in the first argument, be aware that these initializations cannot depend on other variables declared as local variables in the same initialization part of `Module`. Thus, in the following example, `var2` will not have the value of the just-initialized `var1`, because the initialized symbol is actually *var*`$`*number*.

```
Module[{var1 = 2 2, var2 = var1}, {var1, var2}]
```

Also, multiple assignments `{`var_1`,` var_2`,` …`,` var_n`}` `=` `{`$value_1$`,` $value_2$`,` …`,` $value_n$`}` do not work inside `Block`, `Module`, or `With`. Each local variable must be a symbol.

```
Module[{{x, y} = {1, 2}, z[2] = 2}, {x, y, z[2]}]
```

The values of system variables such as `$IterationLimit` or `$RecursionLimit` can also be initialized in `Block`. The current (default) value of `$RecursionLimit` is 256.

```
$RecursionLimit = 256;
```

We now look at defining a function inside of a `Block` or a `Module`. (Note that not only "arguments" but also "heads" get the `$` symbol.)

```
Module[{x, f}, f[x]]
```

To compute `f[258]` in the following procedure, `f` has to be called more than 256 times.

```
Module[{x, f}, f[0] = 0; f[x_] := f[x - 1] + x; f[258]]
```

If we make `$RecursionLimit` a local variable inside a `Block`, we can give it a larger value locally, and thus avoid the error message `$RecursionLimit::reclim`. (Very recursive calculations should generally be put inside a `Block` with appropriately changed `$RecursionLimit`.)

```
Block[{x, f}, f[0] = 0; f[x_] := f[x - 1] + x; f[258]]
```

```
Block[{x, f, $RecursionLimit = 300},
      f[0] = 0; f[x_] := f[x - 1] + x; f[258]]
```

After `Block` is finished, `$RecursionLimit` has its old value.

```
$RecursionLimit
```

The analogous construction does not work with `Module`, because the local variable `$RecursionLimit$`*number* is assigned the value 300, not `$RecursionLimit`.

```
Module[{x, f, $RecursionLimit = 300},
       f[0] = 0; f[x_] := f[x - 1] + x; f[258]]
```

The following nested version of `Block` and `Module` does of course also work.

```
Block[{$RecursionLimit = 300},
Module[{x, f}, f[0] = 0; f[x_] := f[x - 1] + x; f[258]]]
```

In the next input, the `Sin` function is redefined inside the `Block`.

```
Block[{Sin = Cos}, Sin[Pi]]
```

Because attributes are not related to "values", they work also when the attributes are localized inside `Block`. Here is an example.

```
Block[{Orderless, 𝔽, 𝔾},
      SetAttributes[𝔽, Orderless]; SetAttributes[𝔾, Flat];
      {𝔽[2, 1], 𝔾[𝔾[1]]}]
```

The two functions `𝔽` and `𝔾`, however, were local to the `Block` and do not have attributes outside of `Block`.

```
{Attributes[𝔽], Attributes[𝔾]}
```

If a local variable inside a `Module` appears at the same time as a local dummy variable in a scoping construct, these occurrences are not replaced with the renamed variables. This is demonstrated here. The second element in the following list shows a way to circumvent the nonuse of `x$`*number*. We will discuss the meaning of `->` in the next chapter.

```
Module[{t, set}, {Hold[Set[c[t_], t^2]],
                  Hold[set[c[t_], t^2]],
                  Hold[set[c[t_], t^2]] /. set -> Set}]
```

Using `Set` instead of `SetDelayed` yields a similar result.

```
Module[{t, set}, {Hold[SetDelayed[c[t_], t^2]],
                  Hold[setDelayed[c[t_], t^2]],
                  Hold[setDelayed[c[t_], t^2]] /.
                         setDelayed -> SetDelayed}]
```

Because `Block` does not rename the local variables, nothing can go wrong.

```
Block[{t, set}, {Hold[Set[c[t_], t^2]]}]
```

The same behavior holds for `With`.

```
With[{t = Unique["t"], set = Unique["set"]},
     {Hold[Set[c[t_], t^2]],
      Hold[set[c[t_], t^2]] /. set -> Set} ]
```

Here is the renaming of `Pattern[x,_]` within `Block`, `Module`, and `With`.

```
Block[{x = 1}, x_]
```

```
Module[{x = 1}, x_]
```

```
With[{x = 1}, x_]
```

The following “iterative” assignment of values to variables further illustrates the differences between `Block` on the one hand, and `Module` and `With` on the other. To better observe the internal variables, we print them out using `Print[Hold[...]]`.

```
Clear[x, y, z];
```

```
Block[{x = y, y = z, z = 3},
      Print[{Hold[x], Hold[y], Hold[z]}]; {x, y, z}]
```

```
Module[{x = y, y = z, z = 3},
       Print[{Hold[x], Hold[y], Hold[z]}]; {x, y, z}]
```

```
With[{x = y, y = z, z = 3},
     Print[{Hold[x], Hold[y], Hold[z]}]; {x, y, z}]
```

For their iteration variables, `Do`, `Sum`, `Product`, and `Table` use a construction analogous to that of `Block`.

```
i = 3333;
Do[i = i + 1; Print[i], {i, 0, 4}];
```

```
i
```

It might appear that a construction like `Module` would be better, but often a named lengthy expression has to be computed before using `Do`, `Sum`, `Product`, or `Table`. (In addition, `Do`, `Sum`, `Product`, or `Table` allow nonsymbols as iterator variable and try to handle their scoping in a similar manner.)

```
Clear[i];
expression = i^0 + i^1 + i^2 + i^3 + i^4 + i^5 + i^6 + i^7 + i^8
```

If we insert `expression` in `Sum`, we usually get “what we want”.

```
Sum[expression, {i, 1, 10}]
```

If the iteration variables were renamed, we would get the trivial result of 10 times the expression to be summed.

```
Module[{i}, Sum[expression, {i, 1, 10}]]
```

We could again look at this in more detail using `On[]`.

```
Module[{i}, Sum[expression, {i, 1, 2}]]
```

When dealing with nested pure functions, it is often necessary to rename the variables. This is done in a conservative way following the principle “rather one too many, than one too few”. The following function `fufufu` is nested threefold.

```
fufufu = Function[{x}, Function[{y}, Function[{z}, x + y + z]]]
```

If we apply it to a variable `a`, we are left with a function containing within its body another function with head `Func`⋱ `tion`.

```
fufufu[a]
```

The renaming of `y` to `y$` and `z` to `z$` would have been necessary if we had asked for `fufufu[y]` instead of `fufufu[a]`.

```
fufufu[y]
```

We now give a “second” and a “third” argument to `fufufu`.

```
fufufu[y][z]

fufufu[y][z][x]
```

Here, the renaming for an evaluated argument and an unevaluated body of `Function` inside `Block`, `Module`, and `With` is shown.

```
Block[{x = y}, Function[Evaluate[x], x]]

Module[{x = y}, Function[Evaluate[x], x]]

With[{x = y}, Function[Evaluate[x], x]]
```

Be aware of a slightly different scoping behavior of the one-argument pure function compared with its two-argument form. The `Slot` in `#` will not get renamed.

```
Function[Module[{Slot = 1}, Slot[1]]][2]

Function[Slot, Module[{Slot = 1}, Slot[1]]][2]
```

Here is another example involving `Module`. First, the local variable `x$`*number* inside the first argument of `Module` is assigned the value `x^2/2` (without `$`), and it is then output as the evaluation result of the body of `Module`.

```
Clear[x];
Module[{x = Integrate[x, x]}, x]
```

We turn now to `With`: `With` “only” replaces quantities, and it does not create new variables that can be assigned values. Hence, the following construction using `With` does not work.

```
(* comparison with Module *)
Module[{x = 1}, x = 2]

With[{x = 1}, x = 2]
```

Every appearance of the local variable is immediately replaced.

```
With[{x = t + b}, x^2]
```

Even using `Hold`, `ToString`, `Unevaluated`, `HoldPattern`, or `HoldComplete`, it is not possible to get the “variable” `x`.

```
With[{x = t + b},
     {Print[Hold[x]], ToString[x], Unevaluated[x],
      HoldPattern[x], HoldComplete[x]}] // InputForm
```

Here is the only possible way to get `x` in “pure” form.

```
With[{x = t + b}, ToHeldExpression["x"]]
```

It works, in this case, because the variable `x` is not present in the body of the `With`, only the string `"x"` is. However, if a function definition with `Blank` appears inside a `With`, the variables used are of course bound to the function definition.

```
Clear[f, g, x, y]
With[{x = y}, f[x_] := x; g[x_] = x; ]
```

```
??f
```

```
??g
```

Of course, the fact that `Module` creates new variables has an effect on speed of computations compared with `Block`. Here is a long list of variables.

```
Clear["x*"];
```

```
localVars = Table[ToExpression["x" <> ToString[i]], {i, 100}];
```

```
Short[localVars, 5]
```

Now, we use this variable list in `Block` and `Module`, respectively. `Evaluate[localVars]` is necessary because `Block` and `Module` both carry the attribute `HoldAll`. The squaring (i.e., `Power`) is `Listable`. (To have a measurable timing for evaluation, we use a `Do` loop.)

```
Timing[Do[Block[Evaluate[localVars], localVars^2], {2000}]]
```

```
Timing[Do[Module[Evaluate[localVars], localVars^2], {2000}]]
```

If possible, variables should be assigned values in the first argument of `Block`. First, these assignments improve the readability of the program, and second, this is slightly faster than is a value assignment in the second argument. Because we cannot measure very small time intervals with `Timing`, we use `Do[…, {100}]` to get a larger time interval.

```
Timing[Do[
Block[{x1 = 1, x2 = 2, x3 = 3, x4 = 4, x5 = 5, x6 = 6,
       x7 = 7, x8 = 8, x9 = 9, x10 = 10, x11 = 11, x12 = 12,
       x13 = 13, x14 = 14, x15 = 15, x16 = 16, x17 = 17,
       x18 = 18, x19 = 19, x20 = 20}, Null], {10^4}]]
```

The last input is easier to read and faster than is what follows.

```
Timing[Do[
Block[{x1, x2, x3, x4, x5, x6, x7, x8, x9, x10,
       x11, x12, x13, x14, x15, x16, x17, x18, x19, x20},
      x1 = 1; x2 = 2; x3 = 3; x4 = 4; x5 = 5; x6 = 6;
      x7 = 7; x8 = 8; x9 = 9; x10 = 10; x11 = 11; x12 = 12;
      x13 = 13; x14 = 14; x15 = 15; x16 = 16; x17 = 17;
      x18 = 18; x19 = 19; x20 = 20; Null], {10^4}]]
```

`With` protects just as well as `Module`, but it is clearly faster because no new variables have to be created. It is also faster than is `Block`.

```
Timing[Do[
Module[{x1 = 1, x2 = 2, x3 = 3, x4 = 4, x5 = 5, x6 = 6,
        x7 = 7, x8 = 8, x9 = 9, x10 = 10, x11 = 11, x12 = 12,
        x13 = 13, x14 = 14, x15 = 15, x16 = 16, x17 = 17,
        x18 = 18, x19 = 19, x20 = 20}, Null], {10^4}]]
```

```
Timing[Do[
With[{x1 = 1, x2 = 2, x3 = 3, x4 = 4, x5 = 5, x6 = 6,
      x7 = 7, x8 = 8, x9 = 9, x10 = 10, x11 = 11, x12 = 12,
      x13 = 13, x14 = 14, x15 = 15, x16 = 16, x17 = 17,
      x18 = 18, x19 = 19, x20 = 20}, Null], {10^4}]]
```

In addition to faster execution, another reason exists for using `With` instead of `Module` when possible: Because the corresponding variables are assigned values at the outset (values that stay fixed within the scope of `With`), the readability of the program is improved.

Note again that the variables in the first argument of `Block`, `Module`, and `With` must have the head `Symbol`; that is, composite quantities are not allowed.

```
Clear[x];

Block[{x[1] = 1}, x[1]^2]
```

Similar messages would be the result of `Module[x[1] = 1, x[1]^2]` and `With[x[1] = 1, x[1]^2]`. Now, we give a few examples involving local variables.

In the next input, the unbound in (`Function`) variable gets the local variable from `Module`.

```
Module[{x}, Function[y, x + y]]
```

The bound in (`Function`) variable is not replaced by `y$`*number*.

```
Module[{y}, Function[y, x + y]]
```

The following result does not contain `z`, because the inner local variable `x` is not replaced with the value `z` from the enclosing `With`.

```
With[{x = z}, Module[{x}, x + y]]
```

Patterns of the form *var*_ restrict *var* locally to the inside of the associated `Set` or `SetDelayed` in `Module`.

```
Remove[f, x, y, a];
Module[{x = y}, f[x_] = {x}; Print[Definition[f]]; {f[x], f[a]}]
```

Without `Module`, the result looks different.

```
Remove[f, x, y, a];
x = y; f[x_] = {x}; {f[x], f[a]}
```

For comparison, we give a few similar constructions with `Block` and `With`. Here is `Block`.

```
Clear[f, g, x, y, a, b];
Block[{f, g, x, y}, f[x_] = x^2; g[y_] := y^3;
      Print["The definition of f:", Definition[f]];
      Print["The definition of g:", Definition[g]];
          {f[a], g[b]}]
```

Here is `Module`, first without an assignment of values to the local variables.

```
Module[{f, g, x, y}, f[x_] = x^2; g[y_] := y^3;
       Print["The definition of f:", Definition[f]];
       Print["The definition of g:", Definition[g]];
       {f[a], g[b]}]
```

And here is `Module` with an assignment of values to the local variables.

```
Module[{f, g, x = 1, y = 1}, f[x_] = x^2; g[y_] := y^3;
       Print["The definition of f:", Definition[f]];
       Print["The definition of g:", Definition[g]];
       {f[a], g[b]}]
```

In the following construction, `x` in the right-hand side of the definition of `f` is not a variable local to `Set`, and the definition `x = 1` goes inside of `Module`.

```
x = 1; y = 1;
Module[{f, g}, f[x_] = x^2; g[y_] := y^3;
       Print["The definition of f:", Definition[f]];
       Print["The definition of g:", Definition[g]];
       {f[a], g[b]}]
```

Next, we also assign values to the functions in the first argument of `Module`.

```
Clear[f, 𝑓, g, 𝑔, x, y, a, b];
Module[{f = 𝑓, g = 𝑔, x = x1, y = y1},
       f[x_] = x^2; g[y_] := y^3;
       Print["The definition of f:", Definition[f]];
       Print["The definition of g:", Definition[g]];
       {f[a], g[b]}]
```

We now define a function of two variables in `Module`, the first argument as a pattern, and the second argument directly as a specific fixed symbol. Without assigning a value to the local argument in `Module`, we have the following result. The right-hand side of the definition of `f` is bound to the pattern variable `x_`.

```
Remove[𝕗, x, y, a];

Module[{x}, 𝕗[x_, x] := {x, x};
       Print[Definition[𝕗]];
       {𝕗[y, y], 𝕗[y, x]}]
```

Here is an example with a value assignment to the local arguments in `Module`.

```
Remove[𝕗, x, y, z, a];

Module[{x = z}, 𝕗[x_, x] := {x, x};
       Print[Definition[𝕗]];
       {𝕗[y, y], 𝕗[y, x], 𝕗[x, z], 𝕗[y, z]}]
```

In pure functions, the variables are also “internally” localized.

```
Module[{l}, Function[l, l^2]]

With[{l = p}, Function[l, l^2]]
```

The following two examples show how safe the renaming of variables is, in general. However, variables should not be given the same names as system variables, as is done in the following example.

```
quitFunc[Exit_] := Exit^2

quitFunc[5]
```

When we avoid the evaluation of the argument of `quitFunc` (say, by calling it with an unevaluated argument), we can call `quitFunc` in a safe way with the argument `Exit`.

```
quitFunc[Unevaluated[Exit]]
```

The following example also works (although it is not the most recommended use of `Exit`).

```
quitModule[Exit_] := Module[{I = Exit}, Print[I^3]]
```

```
quitModule[3]
```

Inside `Block`, variables have local values. For instance, they can be cleared inside `Block`.

```
x = 1;
Block[{x = 2}, Clear[x]; ; Print[{ToExpression["x"], x}]]
```

Here, the same is done in `Module`.

```
Module[{x = 2}, Clear[x]; Print[{ToExpression["x"], x}]]
```

`With` does not allow us to "clear" local constants.

```
With[{x = 2}, Clear[x]; Print[{ToExpression["x"], x}]]
```

When `x` has a symbolic value, we can remove the corresponding value.

```
With[{x = ZℤℨZ}, Remove[x]; Print[{ToExpression["x"], x}]];
```

To conclude this subsection, we give two examples involving the protection of variables and the interaction of `Block`, `Module`, and `With`. Here is a multiple nesting of the three commands. We encourage the reader to think about what the result might be, and note that the variables have been assigned values in the beginning.

```
k = 3; x = 4; l = 5; i = 9;
Do[sum =
   Sum[Module[{x = 1/With[{k = 1/Block[{l = 1/i}, 1/l k]
                       }, 1/k]}, i x k]^2, {i, 2}],
   {100}] // Timing
```

The value of `sum` is 2.

```
sum
```

The evaluation of this input requires a considerable number of renamings, evaluations, and variable protections during its calculation. Using `Trace`, we can monitor them.

```
Trace[Sum[Module[
 {x = 1/With[{k = 1/ Block[{l = 1/i}, 1/l k]}, 1/k]},
               i x k]^2, {i, 2}]]
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

■ 4.6.4 Localization of Variables in Contexts

Every user-defined or system-defined symbol is in a context. A context is identified by *contextName*` (the name *contextName* and the backquote ` are needed). The system symbols are in the context `System``, whereas the user-defined symbols are typically in the context `Global`` (the `Global`` and `System`` are normally not explicitly written in interactive *Mathematica* sessions). Symbols with the same names from different contexts are completely independent of each other.

`Context[`*symbol*`]` gives the `Context` in which *symbol* is defined.

The built-in symbol `Sin` is in the context `System``.

```
Context[Sin]
```

The variable `newVar` will now be created in the `Gobal`` context.

```
Context[newVar]
```

We now create `x` in two different contexts and give them values.

```
cont1`x = 5; cont2`x = 7;
```

Here are the different versions of `x`.

```
{x, Global`x, cont1`x, cont2`x}
```

Here is a list of all `x` that have so far appeared in any context. Most of them come from packages in the `StartUp` directory, which are read at the beginning of a *Mathematica* session.

```
??*`x
```

Variables can be introduced in any context, including the context `System``. Here, `Amy` is introduced into the `Sys⋱ tem`` context.

```
System`Amy

Names["System`Am*"]
```

Contexts can be nested arbitrarily. (The reader might make use of this property in very large programs.)

```
cont1`cont11`cont111`x

Function[{x}, Context[x], {Listable}][
          {cont111`x, cont11`cont111`x, cont1`cont11`cont111`x,
           cont1`cont11`cont111`cont1111`x}]
```

Using the command `Contexts` in the following example, we can see that a context has to be the form *symbol*`, where *symbol* has the head `Symbol`.

```
Hold[s`α] // FullForm

Hold[(s`)α] // FullForm

Hold[s[1]`α] // FullForm

Hold[(s[1]`)α] // FullForm
```

The current context can be determined with `$Context`.

`$Context`
gives the current context.

```
$Context
```

The current context need not be given explicitly in the form *currentContext`symbol*. As mentioned in the beginning of this subsection, it suffices to write `*symbol* or just *symbol*. It is relatively rare that several symbols with the same names but coming from different contexts need to be used simultaneously. Here are all of the symbols appearing in the current context ``Global`.

```
??Global`*
```

The contexts `Global`` and `System`` currently contain many different symbols.

```
Length[Names["Global`*"]]

Length[Names["System`*"]]
```

Currently no name exists simultaneously in both contexts. (We will discuss the function `Intersection` in Chapter 6.)

```
Intersection[Names["Global`*"], Names["System`*"]]
```

Altogether (meaning in all available contexts), many more symbols exist, of course.

```
Length[Names["*`*"]]
```

Some variables have nested contexts.

```
{Length[Names["*`*`*"]], Length[Names["*`*`*`*"]],
 Length[Names["*`*`*`*`*"]], Length[Names["*`*`*`*`*`*"]]}
```

Symbols in the context `Global`` take precedence over symbols with the same name in the (now to-be-created) context `Symbol``. We make two definitions for the function `asd`; one in the `Global`` context and one in the `System`` context.

```
Clear[r, x];

Global`asd[x_] = x;
Symbol`asd[x_] = x^2;
asd[r]
```

If we want to use the definition of `asd` from the context `Symbol``, we have to explicitly specify the context.

```
Symbol`asd[r]
```

The following input calculates how many symbols are presently available in all currently available contexts.

```
contextsAndVariables[] :=
{First[#], Length[#]}& /@ Sort[Split[Sort[
 Flatten[Table[Context /@ Names[StringJoin[Table["*`", {k}]] <> "*"],
               {k, 0, 6}]]]], Length[#1] > Length[#2]&]

theCurrentContextsAndVariables = contextsAndVariables[]
```

The contexts present in a *Mathematica* session depend crucially from the calculations carried out. For efficiency, many contexts and function definitions are loaded only when needed. So trying to evaluate the following integral adds more than 15 context and nearly 5000 symbols from these contexts.

```
Integrate[HypergeometricPFQ[{a, b}, {c, d, e}, z]^z, {z, 0, 1}]

Select[contextsAndVariables[],
       FreeQ[theCurrentContextsAndVariables, #[[1]], {-1}]&]
```

At the beginning of a *Mathematica* session, all of the built-in system commands (context `System``) are available along with some other, platform-dependent commands. In general, this means without `, all symbols from the contexts that are in the `$ContextPath` are available. The context path can be obtained with `$ContextPath`.

`$ContextPath`

gives a list of the contexts that have been read in, and in which new symbols have been officially introduced. If a symbol appears in multiple contexts, and this symbol name is entered without explicit context specification, *Mathematica* chooses the symbol coming from the context that comes first in `$ContextPath`.

`Contexts[]`

gives a list of all contexts that have been read in.

Mathematica packages create their own contexts to help protect the auxiliary variables they employ. Commands defined there, which are not exported, generally remain invisible.

So far in this *Mathematica* session, we have gone through the following contexts (primarily in the start-up process). These are the official contexts that were used.

```
$ContextPath
```

This is a list of all contexts in use until now.

```
Contexts[]
```

Here are the symbols from the context `Internal``.

```
Names["Internal`*"]
```

Some of the functions from undocumented contexts have self-explanatory names and have some occasionally useful functionality. Here is an example from the last output.

```
(* this will issue no messages *)
Internal`DeactivateMessages[0^0 + 0/0 - Sin[1, 2, 3] - 1[2][[3, 4, 5]]]
```

But in general, it is not recommended to use undocumented functions.

We save the number of symbols in the currently visible contexts, which allows us to monitor any changes in the following example, in which we will add new contexts.

```
nBefore = Length[Names["*"]]
```

Next, we load an external package.

```
Needs["NumberTheory`PrimitiveElement`"]
```

We have now loaded the new contexts in which new variables have been introduced. Here is the new `$ContextPath`.

```
$ContextPath
```

In fact, we have read in some other contexts to help implement commands exported from `NumberTheory`PrimitiveElement``, but they are not included in `$ContextPath`.

```
Contexts[]
```

Just one new variable exists, `PrimitiveElement`.

```
Length[Names["*"]] - nBefore
```

In principle, it is also possible to get to the “hidden symbols”. We have to explicitly specify the context.

```
Length[Names["NumberTheory`PrimitiveElement`Private`*"]]
```

Here is a concrete example.

```
Names["NumberTheory`PrimitiveElement`Private`*"][[21]]
```

Often, the information we get with ??*commandFromAPackage* is hard to understand. First, the individual definitions are given, and second, all symbols are given with their usually lengthy context specifications.

```
?? NumberTheory`PrimitiveElement`Private`primel
```

If, in addition to using some variables from other contexts, we want to change the current context, we can use `Begin`.

`Begin["`*newContext*`` ` ``"]`

changes the current context to *newContext*`.

`End[]`

resets the current context to what it was before the last `Begin["`*newContext*`` ` ``"]`.

In the following example, the current context is changed from `Global`` to the newly created context `co``.

```
Context[]
```

```
Begin["co`"]
```

```
varia = 1
```

This context contains the variable `varia`.

```
Names["*`varia"]
```

Now, we end the context `co``.

```
End[]
```

We are back in the context `Global``.

```
Context[]
```

Inside the `Global`` context, other contexts are explicitly shown (with the exception of the `System`` context).

```
Names["*`varia"]
```

The possibility to change the current context with `Begin` is quite useful for looking at some definitions exported from packages. In the `??NumberTheory`PrimitiveElement`Private`primel` example, all variable names contained the context information, which made them hard to read. By switching the context temporarily to `Number⋱ Theory`PrimitiveElement`Private``, the names are given in much shorter form because the context information is not given for the current context and for symbols from the contexts `System`` and `Global``.

```
Begin["NumberTheory`PrimitiveElement`Private`"]
```

```
NumberTheory`PrimitiveElement`Private`primel
```

```
??primel
```

```
End[]
```

Commands that change the context should always stand alone, never inside other commands. This makes the resulting program easier to read. And it makes sure that all contexts and variables get properly created.

In the following thread, we will shortly discuss the creation of new variables in contexts. In the next input, we start with assigning the value 4 to the variable (living in the context `Global``) `aNewVariable`. Then, we change the context to `nc1`` and assign the value 3 to `aNewVariable`. Because the context `Global`` is in the current context path, no new variable `nc1`aNewVariable` is created, but instead the value of the variable `Global`aNewVariable` is changed. To monitor the values and contexts of the variable `aNewVariable`, we use `Print` statements.

```
(aNewVariable = 4;
 (* new context *)
 Begin["nc1`"];
 (* assign value to a symbol *)
 aNewVariable = 3;
 (* print status *)
 Print["The current value of aNewVariable is: ", aNewVariable];
 Print["The current $ContextPath is: ", $ContextPath];
 Print["The list of all variables matching *`aNewVariable is: ",
       Names["*`aNewVariable"]];
 Print["The context of aNewVariable is: ", Context[aNewVariable]];
 Print["The current context is: ", Context[]];
 End[]);
```

Next, we basically repeat the program above, but this time we do not create a variable `bNewVariable` before the context `nc2`` is created. Again, we assign the value 3 to `bNewVariable`. Because the whole input is parsed in the context `Global``, the variable `bNewVariable` was created in the `Global`` context and no new variable `nc2`b`⋱ `NewVariable` is created, but instead the value of the variable `Global`bNewVariable` gets its the value 3.

```
((* new context *)
 Begin["nc2`"];
 (* assign value to a symbol *)
 bNewVariable = 3;
 (* print status *)
 Print["The current value of bNewVariable is: ", bNewVariable];
 Print["The current $ContextPath is: ", $ContextPath];
 Print["The list of all variables matching *`bNewVariable is: ",
       Names["*`bNewVariable"]];
 Print["The context of bNewVariable is: ", Context[bNewVariable]];
 Print["The current context is: ", Context[]];
 End[]);
```

We repeat the last program once more with minor modifications. This time we do not use parentheses around the whole input. We do not create a variable `cNewVariable` before the context `nc3`` is created. We assign the value 3 to `cNewVariable`. Now, the variable `bNewVariable` will be created in the `nc3`` context.

```
(* new context *)
Begin["nc3`"];

(* assign value to a symbol *)
cNewVariable = 3;

(* print status *)
Print["The current value of cNewVariable is: ", cNewVariable]

Print["The current $ContextPath is: ", $ContextPath]

Print["The list of all variables matching *`cNewVariable is: ",
       Names["*`cNewVariable"]]

Print["The context of cNewVariable is: ", Context[cNewVariable]]

Print["The current context is: ", Context[]]

End[];
```

In the next input, we explicitly remove the `Global`` context from the context path. As a result, when the assignment `dNewVariable = 3` gets carried out, *Mathematica* cannot find a variable of the name `dNewVariable`, and so it creates one in the context `nc4``.

```
dNewVariable = 4;
(* new context *)
Begin["nc4`"];
(* remove the Global` context from the $ContextPath *)
$ContextPath = DeleteCases[$ContextPath, "Global`"];
(* repeat steps from above *)
(* assign value to a symbol *)
dNewVariable = 3;
(* print status *)
Print["The current value of dNewVariable is: ", dNewVariable];
Print["The current $ContextPath is: ", $ContextPath];
Print["The list of all variables matching *`dNewVariable is: ",
      Names["*`dNewVariable"]];
Print["The context of dNewVariable is: ", Context[dNewVariable]];
Print["The current context is: ", Context[]];
End[];
$ContextPath = AppendTo[$ContextPath, "Global`"]
```

Now, we have two variables named `dNewVariable`, one in the `Global`` and in the `nc4`` context.

```
dNewVariable

nc4`dNewVariable
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

■ 4.6.5 Contexts and Packages

As already mentioned, packages serve to implement various routines unavailable in the *Mathematica* kernel. A large number of packages are in the standard packages directory. Of course, the user can write and add (or delete) packages. They are written in the *Mathematica* programming language and have a special structure. We will not describe it in much detail, but instead concentrate on the contexts involved. If the reader needs more advice in writing your own package, see [16✱].

We discussed earlier that `Get[`*file*`]` reads in the file *file*. A second, somewhat safer way exists to read in a *Mathematica* package. We will discuss this `Needs` command shortly.

> *Mathematica* packages are typically loaded as follows: <<*AreaOfMathematics*`*SpecialSub*`*ject*` or `Needs["`*AreaOfMathematics*`*SpecialSubject*``"]`

Here are some examples.

```
<<"Calculus`VectorAnalysis`"
<<"Graphics`Graphics3D`"
```

Note the following exception: A more precise path may be necessary in case the package to be loaded is not in its default directory.

In this subsection, we want to examine exactly how the contexts, the context paths, and the protection of variable names change as we run through a package. To this end, we look at the most rudimentary structure of a package.

•••
General Remarks (author, history, ...) in form of Mathematica comments

`BeginPackage["` *name* `` `"]; ``
•••
Information on the functions defined
in the package which are to be exported using

```
function₁::usage = " someText₁"
function₂::usage = " someText₂"
...
Begin["`Private`"];
•••
Definition of the functions function_i to be exported and
definition of auxiliary functions

End[];
•••
EndPackage[];
•••
```

The naming `` `Private` `` is not necessary, but it is a general rule to use it. Also, for example, the context `` System` `` has this subcontext of the same name with the following variables.

```
Names["System`Private`*"] // Short[#, 12]&
```

Now, at each of the places marked with ••• in our sample package, we will carry out these steps:

■ Write where we are.

■ Write the current context.

■ Write the current context path. To reduce the number of contexts printed, we look only for the new ones here.

■ Assign a value for the variable `xHere`.

■ Write the definition of the variables *context*`` ` ``xHere.

■ Write the functions that are currently directly accessible (i.e., without giving their explicit context).

We give our imaginary package the name `WhatsGoingOnWithContexts`. We first define three functions, `ContextTester`, `VariablesTester`, and `FunctionDefinitionsTester` to help us examine the current context, variables, and function definitions (the function `Complement[`*a*`, `*b*`]` gives all elements of *a* that are not in *b*. To avoid introducing variable form contexts that are only to be defined later, we use constructions of the form `Names["*`varName"]` instead of explicitly listing the various variables as symbols.

All printed statements have attached a small circle ∘ in the beginning to make them easier to recognize as such.

```
contextsBefore = Contexts[];

ContextTester[where_] :=
((* print data about the current context state *)
 CellPrint[Cell[TextData[StyleBox["∘ We are currently here: " <> where,
                   FontWeight -> "Bold"]], "PrintText"]];
 CellPrint[Cell[TextData[{"∘ The value of ",
                  StyleBox["$Context", "MR"], " is: ",
                  StyleBox[ToString[InputForm[$Context]], "MR"]}],
                 "PrintText"]];
 CellPrint[Cell[TextData[{"∘ The value of ",
                               StyleBox["$ContextPath", "MR"], " is: ",
                  StyleBox[ToString[InputForm[$ContextPath]], "MR"]}],
                 "PrintText"]];
 CellPrint[Cell[TextData[{"∘ The new contexts are: ",
                  StyleBox[ToString[InputForm[
                    Complement[Contexts[], Global`contextsBefore]]], "MR"]}]
                 "PrintText"]];)
```

```
VariablesTester :=
 ((* print data about the current state of variables *)
  CellPrint[Cell[TextData[{"∘ The current names of the form ",
                StyleBox["xHere*", "MR"], " are: ",
                StyleBox[ToString[InputForm[Names["xHere*"]]], "MR"]}],
              "PrintText"]];
CellPrint[Cell[TextData[{"∘ The current names of the form ",
                StyleBox["*`xHere*", "MR"], " are: ",
                StyleBox[ToString[InputForm[Names["*`xHere*"]]], "MR"]}],
              "PrintText"]];
CellPrint[Cell[TextData[{"∘ The definition of all ",
                 StyleBox["*`xHere*", "MR"], ": "}], "PrintText"]];
 (CellPrint[Cell[TextData[{"∘ The definition of ",
               StyleBox[#, "MR"], " from context ",
               StyleBox[Context[#], "MR"], " is: "}], "PrintText"]];
  Print[Definition[#]])& /@ Names["*`xHere"];)

FunctionDefinitionsTester :=
  ((* print data about the current state of functions *)
   CellPrint[Cell[TextData[{"∘ The current names of the form ",
              StyleBox["our*", "MR"], " are: ",
              StyleBox[ToString[InputForm[Names["our**"]]], "MR"]}],
            "PrintText"]];
   CellPrint[Cell[TextData[{"∘ The current names of the form ",
              StyleBox["*`our*", "MR"], " are: ",
              StyleBox[ToString[InputForm[Names["*`our**"]]], "MR"]}],
            "PrintText"]];
   CellPrint[Cell[TextData[{"∘ The definition of all ",
                  StyleBox["`our*", "MR"], " : "}],
                "PrintText"]];
   (CellPrint[Cell[TextData[{"∘ The definition of ",
                   StyleBox[#, "MR"], " from context ",
                   StyleBox[Context[#], "MR"], " is: "}],
                "PrintText"]];
  Print[Definition[#]])& /@ Names["*`our*"];)
```

Here is the outline of our (toy-)package and the information about context changes and variable assignments during its evaluation. Note the introduction of the function names `ContextTester`, `VariablesTester`, and `FunctionDefinitionsTest` using the context `Global``. (Inside the current context, only commands from that context or from the context `System`` can be used without explicitly giving the context specification.) The various commands are all "single", that is, no semicolons exists.

This is the state of the contexts, before any change, related to imitating a package.

```
Global`ContextTester["Before BeginPackage"]

xHere = beforeBeginPackage

Global`VariablesTester

Global`FunctionDefinitionsTester
```

The `BeginPackage` changes the context to `WhatsGoingOnWithContexts``. This current context is not in the list of the contexts returned by `Contexts[]`.

```
BeginPackage["WhatsGoingOnWithContexts`"]

Global`ContextTester["After BeginPackage"]
```

Note that the context `WhatsGoingOnWithContexts`` is not in the list of the contexts returned by `Contexts[]`

(because we are just walking through it).

```
Contexts[]
```

Now, we define a `xHere` in the current context. The commands to be exported from a package are also introduced at this point (see below), and the names used in the context `Global`` are still visible. So using the name `xHere` at this point results in a warning message.

```
xHere = afterBeginPackage

Global`VariablesTester
```

We do not define the function `ourFunction` explicitly at this point, but introduce its symbol and give a usage message here.

```
ourFunction ::usage = "ourFunction forms twice the square of a number"

Global`FunctionDefinitionsTester
```

Now, we switch to the subcontext ``Private`` of the context `WhatsGoingOnWithContexts``.

```
Begin["`Private`"]
```

Now, after we left the context `WhatsGoingOnWithContexts``, it appears in the result of `Contexts`.

```
Global`ContextTester["After BeginPrivate"]
```

Again, we define a variable `xHere`. Because a variable with the name `xHere` already exists in the available contexts (in `WhatsGoingOnWithContexts``), the value of `WhatsGoingOnWithContexts`xHere` is overwritten and no new variable `WhatsGoingOnWithContexts`Private`xHere` is created.

```
xHere = afterBeginPrivate

Global`VariablesTester
```

Inside this innermost context of a typical package, we define the function to be exported (here, `ourFunction`) and some auxiliary functions that are needed to define it.

```
ourAuxiliaryFunction[x_] := x^2

ourFunction[x_] := 2 ourAuxiliaryFunction[x]
```

At this point, both functions `ourFunction` and `ourAuxiliaryFunction` are visible and match the pattern `"our*"` without giving explicit context specifications in the variable name.

```
Global`FunctionDefinitionsTester
```

The functions exported from a package are typically protected.

```
Protect[ourFunction]
```

The next `End[]` ends the context ``Private``, and after this, we are back in the context `WhatsGoingOnWithCon`↵ `texts``.

```
End[]

Global`ContextTester["After End"]
```

Again, we define a variable `xHere`. (In a typical package, nothing is defined at this place.)

```
xHere = afterPrivate

Global`VariablesTester

Global`FunctionDefinitionsTester
```

The `EndPackage[]` now ends the context `WhatsGoingOnWithContexts``.

```
EndPackage[]
```

Now, we have gone through our whole package and we are back in the context `Global``, and the package context `WhatsGoingOnWithContexts`` is now part of the context path. The subcontext `WhatsGoingOnWithContexts`Private` is not in the context path.

```
Global`ContextTester["After EndPackage"]
```

Again, we give the variable `xHere` a value.

```
xHere = afterPackage
```

```
Global`VariablesTester
```

The function `ourFunction` is available now also in the current context.

```
Global`FunctionDefinitionsTester
```

Now, we are finished going through all the steps of context changes in a package. The function `ourFunction` is now available for use.

```
ourFunction[abc]
```

But the function `ourAuxiliaryFunction` is not known in the current context.

```
ourAuxiliaryFunction[abc]
```

We can access the definition of `ourAuxiliaryFunction` by calling `ourAuxiliaryFunction` with its context specification.

```
WhatsGoingOnWithContexts`Private`ourAuxiliaryFunction[abc]
```

We note the following concerning the changes in the contexts:

■ After `BeginPackage`, `$ContextPath` contains only the newly created context `WhatsGoingOnWithContexts` and `System`.

■ ``Begin["`Private`"]`` does not change the `$ContextPath`.

■ In ``Begin["`Private`"]``, the `` `Private` `` has a ` on the left; that is, it is a subcontext of `WhatsGoingOnWithContexts`.

■ After `End[]`, the context `` WhatsGoingOnWithContexts`Private` `` is not in `$ContextPath`. Thus, commands defined there cannot be called without giving the explicit context.

■ The function exported lives in the context specified by `BeginPackage[`*context*`]`.

```
Context[ourFunction]
```

■ The function `ourFunction` is also directly accessible inside of the context `` WhatsGoingOnWithContexts`Private` ``, which allows us to implement rules for it at this place.

The exported functions of a package often carry the attribute `Protected`, which necessarily causes problems if a package is read in more than once, because functions that were already named are defined again. Here is an example from the standard packages exhibiting the problem.

```
<< NumericalMath`Approximations`
```

```
<< NumericalMath`Approximations`
```

This problem can be avoided with `Needs`, which looks at the `$ContextPath` if the package was already loaded.

```
$ContextPath
```

```
Needs["NumericalMath`Approximations`"]
```

`Needs["`*context`string*`"]`

reads in the file that is associated with the context *context`string* (as a string), provided this package has not yet been read.

Here, we read in the package `NumberTheory`NumberTheoryFunctions``.

```
Needs["NumberTheory`NumberTheoryFunctions`"]
```

It includes, for example, the function `SumOfSquaresR`, which counts the number of ways to represent an integer *n* as a sum of *d* squares.

```
?SumOfSquaresR
```

```
Table[SumOfSquaresR[d, 2], {d, 12}]
```

```
Table[SumOfSquaresR[2, n], {n, 12}]
```

The following command does not read in the package again.

```
Needs["NumberTheory`NumberTheoryFunctions`"]
```

In addition to the problem caused by loading a package more than once, another problem can also arise: A package may contain a function with the same name as one we have already used, but with a different definition. Here, we define a very naive function `ContourPlot3D` in the current context `Global``.

The same function is contained in the package `Graphics`ContourPlot3D``.

```
ContourPlot3D[func_, xIter_, yIter_, zIter_] :=
    Show[Table[Graphics3D[(* 2D contour plot *)
              Graphics[ContourPlot[func, xIter, yIter,
                          ContourShading -> False,
                          ColorFunctionScaling -> False,
                          Contours -> Table[c, {c, 0, 1, 1/15}],
                          ContourStyle -> Table[{Thickness[0.001],
                                        Hue[h]}, {h, 0, 0.8, 0.8/15}],
                          DisplayFunction -> Identity]][[1]]] /.
     (* lift lines into 3D *)
     Line[l_] :> Line[Append[#, z]& /@ l],
     Evaluate[Append[zIter, (zIter[[3]] - zIter[[2]]) /15]]],
                  DisplayFunction -> $DisplayFunction]
```

```
ContourPlot3D[x^2 + y^2 + z^2 - 1, {x, -1, 1}, {y, -1, 1}, {z, -1, 0}]
```

The same function is contained in the package `Graphics`ContourPlo3D``.

```
Needs["Graphics`ContourPlot3D`"]
```

Because the function `ContourPlot3D` is intended to be exported from the context `Graphics`ContourPlot3D``, a conflict may appear with the command with the same name that was already defined in the context `Global``. The definition that was made first and that appears first in `$ContextPath` remains in effect; that is, the command which has been read in is not recognized.

```
?ContourPlot3D
```

To get the latter definition, we have to specify its context explicitly.

```
?Graphics`ContourPlot3D`ContourPlot3D
```

Using `Graphics`ContourPlot3D`ContourPlot3D`, we obtain a different graphic.

```
Graphics`ContourPlot3D`ContourPlot3D[
     x^2 + y^2 + z^2 - 1, {x, -1, 1}, {y, -1, 1}, {z, -1, 0}]
```

For further details on contexts and packages, see [16✱], Chapters 1 and 2. As mentioned already in the preface, we will not further discuss the design of packages here. See also *MathSource* 0204-961.

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

■ 4.6.6 Special Contexts and Packages

In a typical *Mathematica* session, variables exist in many contexts. Many *Mathematica* functions are written in the *Mathematica* language, and the code for supporting these functions exists in certain specialized contexts. Many of the built-in functions reside in the start-up packages; many other functions reside in the standard packages. Here is the current list of the contexts.

```
Contexts[]
```

In the last result, we see the context `Integrate``, where most of the code for indefinite and definite integration of special functions exists. We also see the context `SymbolicSum``, where the code for symbolic summation exists. And we see many more contexts. Among all of the contexts from the list above, besides the contexts `Global`` and `System``, three other contexts are especially important. The first one is the context `Developer``, which contains more advanced mathematical and programming functions. These functions are typically not needed by a beginning *Mathematica* user, but by experienced users. Here is a complete list of the function names from the `Developer`` context.

```
Names["Developer`*"]
```

We will not go through these functions at this point in detail here, but just having a short glance at what exists in this context might be useful. (We will discuss some of them in the following chapters. At the place, where they belong according to their functionality.) One group of functions are specialized simplifiers.

```
Names["Developer`*Simplify"]
```

Although all of *Mathematica*'s simplification power is available in just two functions (`Simplify` and `FullSimplify`), sometimes we want to simplify only certain classes of functions and this as fast as possible. In such a situation, these self-explanatory simplifiers come in handy. Their naming is obvious, `Developer`GammaSimplify` simplifies only Gamma functions, `Developer`PseudoFunctionsSimplify` simplifies only pseudofunctions (`DiracDelta`, `UnitStep`, …), and so on.

A second set of functions operate at the binary representations of numbers. They are called bit operations.

```
Names["Developer`Bit*"]
```

A third set of functions is related to packed arrays. (Roughly speaking, packed arrays are rectangular *d*-dimensional arrays of machine numbers that allow us to carry out purely numerical calculation at a faster speed by bypassing the standard *Mathematica* evaluation process.)

```
Names["Developer`*Packed*"]
```

As we have already seen, many functions in *Mathematica* allow for options to tune their behavior for special purposes. We could imagine that some of *Mathematica*'s function could have more options for further fine-tuning. Such options probably would not be used too often, so having them always around is a bit of ballast. Many of such options influence more than one *Mathematica* function in their behavior and are collected in the so-called system options. Here is a list of the system options and their current settings. (In analogy to SetOptions, the function Developer`SystemOptions allows to set system options.)

```
Developer`SystemOptions[]
```

Note that these system options are not symbols within the Developer` context, but they are strings. The string quotes are invisible in ordinary StandardForm output.

```
InputForm[%] // Short[#, 4]&
```

Many of the system options are related to compilation and autocompilation. Invisible to the user, many functions (see Chapter 1 of the Numerics volume [28✱]) compile or autocompile their arguments. Here, we select all system options related to this hidden as well to explicitly invoked compilation.

```
Select[First /@ Developer`SystemOptions[], StringMatchQ[#, "*Compile*"]&]
```

The current settings of the system options can be changed by the user. The function that changes a system option is Developer`SetSystemOptions[].

The next most important context after System` and Global` and Developer` is the Experimental` context. Similar to the Developer` context, this context contains many functions for advanced work. Sometimes using experimental functions will be very useful, but we must be aware that no guarantee exists that the interface and syntax of these functions will not change in later versions of *Mathematica*.

```
Names["Experimental`*"]
```

A first group of functions from the Developer` context is related to importing and exporting data.

```
Join[Names["Experimental`*Import*"],
     Names["Experimental`*Export*"]]
```

A second group of functions is related to quantifier elimination and cylindrical algebraic decomposition. Here, such functions as Experimental`CylindricalAlgebraicDecomposition, Experimental`GenericCylindricalAlgebraicDecomposition, Experimental`ImpliesQ, Experimental`ImpliesRealQ, and others belong. (We will discuss many of these functions in Chapter 1 of the Symbolics volume [29✱].)

A further context of interest is the context FrontEnd`.

```
Names["FrontEnd`*"]
```

Because this book does not deal with front end programming, we will not discuss these functions.

The last context to be mentioned here is the context Internal`. The advanced user might find it interesting to experiment with some of the functions from this context, but similar to the functions from the Experimental` context, no guarantee exists that the behavior and syntax of these functions will still be available in later versions of *Mathematica*.

```
Names["Internal`*"]
```

To be efficient in the memory usage, *Mathematica* has only some standard as well as the currently necessary specialized code in memory, and the use of further specialized functions will result in loading relevant code. If this *Mathematica* session was started at the beginning of this subsection, about 3000 "official" symbols (in all contexts) are present and about 1 MB of memory is in use.

```
allCurrentOfficialNames =
Join[Names["System`*"], Names["Developer`*"], Names["Experimental`*"]];
          (* if you want to experiment
       for the brave only ☺)  add : Names["System`*"] *)

(* number of official names and number of all names *)
{Length[allCurrentOfficialNames], Length[Names["*`*"]]}
```

```
(* memory usage and context number *)
{MemoryInUse[], Length[Contexts[]]}
```

We force the autoloading of all functions by converting the string "*symbolName*`[]`" into an expression for all currently available symbols *symbolName*.

```
(* force autoloading by use of function[] *)
((* watch progress: Print[#]; *)
  Block[{(* avoid printed messages *) $Messages = {}},
       ToExpression[# <> "[]"]])& /@
(* delete some "dangerous" functions
  (functions when one called with zero arguments
   expect some additional input) *)
DeleteCases[allCurrentOfficialNames,
             "Abort" | "Break" | "Continue" | "Dialog" | "Exit" |
             "Quit" | "ExitDialog" | "Edit" | "EditDefinition" |
             "EditIn" |  "Print" | "ConsolePrint" | "On" |
             "Input" | "InputString" | "$Inspector" |
             "FileBrowse" | "Experimental`FileBrowse" |
             "Experimental`FindTimesCrossoverDigits" |
             "Internal`FromDistributedTermsList" |
             "InputString" | "NotebookCreate" | "Interrupt" |
             "NotebookOpen" | "NotebookPut" | "ConsoleMessage" |
             "NotebookGet" | "NotebookSave"];
```

Now many more symbols from many more contexts are now present (and considerably more memory are in use).

```
(* number of official names and number of all names *)
{Length[allCurrentOfficialNames], Length[Names["*`*"]]}
```

```
(* memory usage and context number *)
{MemoryInUse[], Length[Contexts[]]}
```

At this point, we should say something about the contents of packages. Over 200 packages are distributed with *Mathematica*. The easiest way to maintain an overview of what has been implemented in these packages is with the use of the help browser. Here, we will look into a more program-oriented approach for getting such an overview. The package `Utilities`Package`` is available on all machines. It contains some metapackage commands.

```
Needs["Utilities`Package`"]
```

Here, we are interested only in the command `Annotation`.

```
?Annotation
```

Using the command `Annotation`, we can implement the command `PackageContents`, which gives either a short (if the second argument of `PackageContents` is `short`) or a detailed (if the argument is `long`) description of the package.

```
SetAttributes[PackageContents, Listable]

PackageContents[package_, length_:short] :=
((* print a header line *)
 CellPrint[Cell[TextData[{"∘ An Overview over the package ",
         StyleBox[package, "MR", FontWeight -> "Bold"], ":"}],
                 "PrintText"]];
 (* print the information *)
 If[length === short,
   Print[package, Annotation[package,
                 {"Name", "Title", "Summary", "Limitations"}]],
   Print[package, Annotation[package,
         {"Copyright", "Mathematica Version", "Package Version",
          "Name", "Title", "Author", "Keywords", "Requirements",
          "Warnings", "Sources", "Summary", "Limitations",
          "Examples"}]]];)
```

Here is an example. We use the standard package `Algebra`AlgebraicInequalities``.

```
PackageContents["Algebra`AlgebraicInequalities`", Long]
```

We can now analyze the package `ChapterOverview`, which we have been using at the end of every chapter.

```
AppendTo[$Path, StringDrop[
  ToFileName["FileName" /. NotebookInformation[SelectedNotebook[]]], -5]];
```

```
PackageContents["ChapterOverview`", long]
```

Because we have given `PackageContents` the attribute `Listable`, we can get all available information on all available packages with the following few lines. (Because of space limitations, we do not execute the next input here.)

```
PackageContents[
  FileNames["*.m", {directorySpecificationForMathematicaPackages}, Infinity], long]
```

The various packages contain a great many commands (more than the number of built-in commands). If we need to work with a large number of these commands at one time, we can read in the so-called master packages through the context of the mathematical subject. They cover one entire mathematical or application subject and contain all *Mathematica* commands from the corresponding directory of packages. When a command listed in the master package (with attribute `Stub`) is used for the first time, the appropriate package is loaded using `Needs`. If the command is only used as a string, no package is loaded, but as soon as it is used explicitly (meaning evaluated), for example, in `ToHeldExpression["`*packageCommand*`"]`, it is loaded. This process saves us from having to read in the individual packages and, moreover, saves memory because only the necessary packages are loaded at any given point. We now look at the commands in the master packages. Because we will count symbols in the following inputs, we recommend restarting *Mathematica* here.

`Off[DeclarePackage::aldec]` prevents the printing of messages in case some of the following master packages were already loaded in the start-up process.

```
Off[DeclarePackage::aldec];
before = Names["*"];
```

Now, we look at the various subjects. (`Complement[`*a*`, `*b*`]` gives a list of everything that is in *a*, but not in *b*; see Chapter 6 for details.)

Here is one for algebra.

```
Needs["Algebra`"]
```

```
newAlgebra = Complement[Names["*"], before];
before = Names["*"]; newAlgebra
```

We only determine how many new commands are defined in the packages. It would be straightforward to list them all, but they would occupy several pages. The first set of commands is for calculus.

```
Needs["Calculus`"]
```

```
newCalculus = Complement[Names["*"], before];
before = Names["*"]; newCalculus // Length
```

More than 300 commands are in the discrete mathematics package.

```
Needs["DiscreteMath`"]
```

```
newDiscrete = Complement[Names["*"], before];
before = Names["*"]; newDiscrete // Length
```

Here is a package for geometry.

```
Needs["Geometry`"]
```

```
newGeometry = Complement[Names["*"], before];
before = Names["*"]; newGeometry // Length
```

The graphics package has about 450 additional commands.

```
Needs["Graphics`"]
```

```
newGraphics = Complement[Names["*"], before];
before = Names["*"]; newGraphics // Length
```

Here is a package for linear algebra.

```
Needs["LinearAlgebra`"]
```

```
newLinear = Complement[Names["*"], before];
before = Names["*"]; newLinear // Length
```

About 750 commands are in the `Miscellaneous`` package.

```
Needs["Miscellaneous`"]
```

```
newMisc = Complement[Names["*"], before];
before = Names["*"]; newMisc // Length
```

This package is about number theory.

```
Needs["NumberTheory`"]
```

```
newNumber = Complement[Names["*"], before];
before = Names["*"]; newNumber // Length
```

Here is one for numerical mathematics.

```
Needs["NumericalMath`"]
```

```
newNumeric = Complement[Names["*"], before];
before = Names["*"]; newNumeric // Length
```

This package is for statistics.

```
Needs["Statistics`"]
```

```
newStat = Complement[Names["*"], before];
before = Names["*"]; newStat // Length
```

Here is a utilities package.

```
Needs["Utilities`"]

newUtilities = Complement[Names["*"], before];
before = Names["*"]; newUtilities // Length

On[DeclarePackage::aldec];
```

Adding all numbers, we have more than 2100 additional commands at our disposal. (`Join` combines the separate lists into one new one; `Sort[…, StringLength[#1] < StringLength[#2]&]` sorts them by length.)

```
allExportedPackageCommands =
   Sort[(* form list of all package functions *)
        Join[newAlgebra, newDiscrete, newCalculus,
             newGeometry, newGraphics, newLinear,
             newMisc, newNumber, newNumeric,
             newStat, newUtilities],
              StringLength[#1] < StringLength[#2]&];

Length[allExportedPackageCommands]
```

Here are the 10 longest exported function names.

```
Take[allExportedPackageCommands, -10]
```

Here is the definition of the function with the longest name.

```
Information[Evaluate[%[[-1]]]]
```

The functions defined in the packages contain many useful functions. The following code measures the size in kB of the full definition of all functions from the list `allExportedPackageCommands`. The graphic shows the cumulative number of functions versus the size of its defining *Mathematica* code.

```
(* delete "dangerous" items *)
allExportedPackageCommands =
DeleteCases[allExportedPackageCommands,
 "FindIons" | "AtomicData" | "AirWavelength" |
 "DampingConstant" | "VacuumWavelength" | "RelativeStrength" |
 "OscillatorStrength" | "ElementAbsorptionMap" |
 "TransitionProbability" | "UpperStatisticalWeight" |
 "LowerStatisticalWeight" | "LowerTermFineStructureEnergy"];

(* unprotect all functions to allow for sub-definitions *)
Unprotect /@ allExportedPackageCommands;

Module[{definitionSizes, function},
definitionSizes =
Table[function = allExportedPackageCommands[[k]];
      ToExpression["Hold[" <> function <> "[]]"];
      (* determine size of definition *)
      ByteCount[ToString[FullDefinition[Evaluate[function]]]],
       {k, Length[allExportedPackageCommands]}];
(* show graphics of logarithm of byte size of definitions *)
ListPlot[Reverse /@ MapIndexed[{#2[[1]], Log[10, #1/1000]}&,
                                  Sort[definitionSizes]],
         PlotRange -> All, Axes -> False, Frame -> True,
         FrameLabel -> {"10^size kB", "number of functions"}]]
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

4.7 The Process of Evaluation

In this section, we discuss the standard procedure used for the computation of a *Mathematica* expression. Knowledge of this procedure is essential for analyzing situations in which things do not go as expected. Here is an example.

First, we define a new head; the head `head8`.

```
head[i_] := ToExpression["head" <> ToString[i]];

SetAttributes[head8, {Orderless, Listable}];
```

We also define a special head.

```
head8[x_, y_] := headache[x, y];
```

Next, we make a small change to `Sin`.

```
Unprotect[Sin];
Sin[x_] = mySin[x];
Protect[Sin];

mySin[x_] := {Sin, x};
```

Here is a list of what we have defined.

```
Definition[head8]

Definition[Sin]
```

Note that with `FullDefinition[`*symbol*`]`, only the definitions of the symbols that appear recursively in the definition of *symbol* without the attribute `Protected` are displayed. Compare the following inputs.

```
FullDefinition[Sin]

Unprotect[Sin]

FullDefinition[Sin]

Protect[Sin];
```

Here is the expression to be computed.

```
head[3 + 5][{Sin[Pi/6], 1}, {2, Cos[Pi/6]}]
```

In view of the following behavior of a command with the attribute `Listable`, the appearance of three elements in the result of `head[3 + 5][{Sin[Pi/6], 1}, {2, Cos[Pi/6]}]` is at first glance rather surprising. Note, on the other hand, the following behavior.

```
SetAttributes[listableFunction, Listable];
listableFunction[{var1pp1, var1pp2}, {var2pp1, var2pp2}]

listableFunction[{{v11, v12}, v2}, {v31, v32}]
```

The form of the results can now be accounted for if we understand the standard procedure for the calculation of an expression (an apostrophe ′ on a symbol means that its value was possibly changed during the computation).

The standard procedure for the evaluation of a *Mathematica* expression of the form *head*[*element*$_1$, *element*$_2$, …] is as follows:

• Compute *head* with the result *head*`.

- Compute *element*$_1$, *element*$_2$,… in the order of their appearance, provided *head* does not carry a `Hold` attribute like `Hold`.
- If *head*\` has one of the attributes `Flat`, `Listable`, or `Orderless`, carry out the resulting transformation.
- If the resulting object has the form *head'*[*subHead*$_1$[*element*$'_1$, *element*$'_2$, …], *subHead*$_2$[…], …], apply the user-defined rules for the entire expression *subHead*$_1$[*element*$'_1$, *element*$'_2$, …], *subHead*$_2$[…], …. Then apply the system rules for the entire expression *subHead*$'_1$[*element*$''_1$, *element*$''_2$, …], *subHead*$'_2$[…], ….
- Apply the user-defined rules for the entire expression *head'*[*element*$'''_1$, *element*$'''_2$, …]. Then, apply the system rules for the entire expression *head''*[*element*$''''_1$, *element*$''''_2$, …].
- Repeat the above steps for any symbol that changed.

With this knowledge of the order in which calculations are carried out, and with the help of `Trace[`*toBeCalculated*`]`, we can now explain what happens for the above example `head[3 + 5][{Sin[Pi/6], 1}, {2, Cos[Pi/6]}]`. The key point is that first the arguments `{Sin[Pi/6], 1}` and `{2, Cos[Pi/6]}` are computed to be `{{Sin, Pi/6}, 1}` and `{2, Sqrt[3]/2}`, and then the attribute `Listable` of `head[8]` goes into effect for these arguments.

```
Trace[head[3 + 5][{Sin[Pi/6], 1}, {2, Cos[Pi/6]}]]
```

Here is a syntactically correct, but semantically not very sensible, expression that shows when the head and when the arguments are calculated.

```
(((Print[a]; a)[(Print[b]; b)])[(Print[c]; c)])[(Print[d]; d)]
```

The following example clearly demonstrates that the `Listable` attribute goes into effect before the actual definitions for *fℓ* are matched.

```
SetAttributes[fℓ, Listable]

fℓ[x_List] := "a list argument"
fℓ[x_] := "any argument"

fℓ[{1, 2, 3}]
```

We now present a somewhat artificial but very useful example to help understand the process of a computation. We begin with a definition and look at how it works.

```
Clear[a, f];

a /: f_[a, b_] := g[f, a, b]

f[a, b]
```

Here is the example program … `/; OrderedQ[{b, c}]` restricts the applicability of the definition of `z` to those cases in which `b` and `c` are in canonical order; we discuss this construction in detail in the next chapter.

```
Clear[f, h, y, z, hi, p, q];

z /: f_[b_, c_, z] := f[hi[b, c], z] /; OrderedQ[{b, c}];
p := b; q := c;
hi[ξ_, η_] := 𝕙𝕚[ξ, η];
SetAttributes[h, Orderless]
h[z, p, q]
```

The order of evaluation of the last expression is as follows. First, the arguments of h are evaluated and the result is h[z, b, c]. Because of the Orderless attribute of h, the arguments in h[z, b, c] become reordered to h[b, c, z]. Now, the upvalue definition for z comes into play and the result is h[hi[b, c], z]. hi[b, c] evaluates to 𝕙𝕚[b, c]. Now again the Orderless attribute of h reorders the arguments of h to h[z, 𝕙𝕚[b, c]]. No further definition applies, and the result is output.

Here is a somewhat different definition.

```
Clear[f, h, y, z, hi, p, q];
ClearAttributes[h, Orderless];

z /: f_[b_Integer, c_Rational, z] := f[hi[b, c], z]
p := 31/11; q := 2;
hi[ξ_, η_] := 𝕙𝕚[ξ, η];
SetAttributes[h, Orderless]
h[z, p, q]
```

Reversing p and q gives the same result.

```
h[z, q, p]
```

With On[], we can clearly follow the order of the calculation. First, the arguments of h, that is, z, p, and q, are computed. Then, the attribute Orderless is applied, and the first hi in h[hi[…], …] is evaluated.

```
On[];  h[z, p, q];  Off[];

Off[]
```

In the case p < q, we get a trivial result despite the attribute Orderless of h, which might be an unexpected behavior.

```
p := 2/3; q := 3;
{h[z, p, q], h[z, q, p]}
```

We now look at a few additional examples to illustrate the order in which *Mathematica* commands are carried out. Here is a structure with a threefold Set.

```
Clear[x, y, z];
On[];
x = y = z = 2

Off[];
```

The order of this computation can be understood if we examine the FullForm of the expression.

```
FullForm[Hold[x = y = z = 2]]
```

It is interesting to look at the same thing with SetDelayed.

```
Clear[x, y, z];
x := y := z := 2;
FullForm[Hold[x := y := z := 2]]

FullDefinition[x]
```

Because x has not yet been called, the right-hand side of the definition of x has not been carried out and no definition has been given for y.

```
??y
```

The result for the evaluation of the variables x, y, and z appears rather puzzling at first glance.

```
{x, y, z}
```

Using `On[]`, here is what happens.

```
??x

FullDefinition[x]

FullDefinition[y]

FullDefinition[z]

On[];
y

Off[];
```

`z` is assigned the value 2, and the variable `y` is assigned the result of the assignment `SetDelayed[z, 2]`, which is `Null`. The same reason also holds for the result `Null` of `x`.

The separate steps of the computation are carried out completely for every substep. The following simple example makes this process clear. First, the first argument of `Level` is evaluated and then the second one is evaluated, with the side effect that an assignment to `a` exists. Then, the actual command is executed, and finally, `a` (which now has the value 2) is evaluated.

```
Clear[a, b, c];

Level[Print["The first argument is being evaluated."];
      {a, b, c},
      Print["The second argument is being evaluated."];
      a = 2; {1}]
```

Using `On[]` we see clearly that the value for `a` was substituted after `Level` was evaluated.

```
Clear[a, b, c];
On[];
Level[{a, b, c}, a = 2; {1}]
Off[]
```

In the next example, we get an empty list as the result, because when `Level` goes into effect, `a` has no nontrivial tree structure.

```
Clear[a, b, c, d];

Level[Print["1st argument is being evaluated "];
      a,
      Print["2nd argument is being evaluated "];
      a = b[c, d]; {1}]
```

Here is a comparison.

```
{Clear[a]; Level[a, {1}], Level[b[c, d], {1}]}
```

In the next input, `ArcTan` has only two remaining arguments at the time it is called, and thus no error message is generated.

```
ArcTan[1, 2, Sequence[]]
```

Arguments are evaluated before the application of `Flat`, `Orderless`, or `OneIdentity`. Thus, the expression `flat[flat[x], flat[x, flat[x]]]` in the following is not reduced to `alsoFlat[x, x, x]`, but to `alsoFlat[alsoFlat[x], alsoFlat[x, alsoFlat[x]]]` (the pattern x___ stands for an arbitrary number of arguments; see the next chapter).

```
Remove[flat, x]
SetAttributes[flat, Flat]
```

```
flat[x___] := alsoFlat[x]

flat[flat[x], flat[x, flat[x]]]
```

However, the following is reduced.

```
Remove[flat, x]
SetAttributes[flat, Flat]

flat[flat[x], flat[x, flat[x]]]
```

While in principle expressions should be evaluated until nothing changes anymore, in practice there are certain limitations and optimizations to this rule to avoid infinite recursions. So in the following example, the first argument T of `Part` is not reevaluated after it got a value when evaluating the second argument of `Part`.

```
T[[T = {1}; 1]]
```

A next complete pass through evaluation gives the expected result `1`.

```
%
```

In the following, similar example, the outer `Set` functions causes a reevaluation.

```
{sin[sin = Sin; 1.],
 Clear[sin];    (* now with outer Set *)
 sinT = sin[sin = Sin; 1.]}
```

Not everything in *Mathematica* is computed according to the above standard procedure. Here are the most important exceptions, allowed primarily to speed up computations and to allow for scoping.

Deviations from the standard procedures for evaluations follow:
- Logical operations are computed only up to the point where their truth value can be uniquely determined.
- Iteration constructions first find the iteration limits and then localize the iteration variables. Values assigned to these variables outside the iteration construction are temporarily ignored.
- Function definitions with `set` or `SetDelayed` calculate the arguments on the left-hand side of the function definition, provided they do not have the head `Symbol`.
- User intervention in the standard calculation procedures is possible using constructions with `Evaluate` and `Unevaluated`, in which the arguments are either computed or not computed, respectively.
- Debugging done with `Trace`.

For a complete discussion of the process of the evaluation of *Mathematica* expressions, including the possible appearances of `Evaluate`, `Unequal`, `Sequence`, or composite heads, see [31✱] and [32✱].

We now look at a graphics example to see the effect of the `Hold` attribute. The following works.

```
Plot[{x, x^2, x^3, x^4}, {x, 0, 1}]
```

Now, we first calculate the functions to be plotted and then draw them.

```
Clear[x];
preComp = {x, x^2, x^3, x^4}
```

Because `preComp` cannot be plotted in “an unprocessed state”, we get an error message (`preComp` gives `{x, x^2, x^3, x^4}` for every inserted value of `x` that is a list, but at the time `x` is inserted in `preComp`, *Mathematica* expects to get a number). At the beginning, the symbol `preComp` is interpreted as one function to be plotted, but later this is not the case.

```
Plot[preComp, {x, 0, 1}];
```

In this case, we have to make sure "by hand" that `preComp` is an object that can be plotted.

```
Plot[Evaluate[preComp], {x, 0, 1}];
```

Here is another example of the meaning of the `Hold` attribute. `Set` possesses the following attributes.

```
Attributes[Set]
```

Why does `Set` carry the attribute `HoldFirst`? We examine the following construction in detail.

```
Clear[f];
f[x_] = x^2;
f[x_] = x^3;
??f
```

At the end of this input, only the second definition is in effect. If `f[x_] = x^3` (i.e., `Set[f[Pattern[x, Blank[]]], Power[x, 3]]` had not been carried out with the attribute `HoldFirst` from `Set`), all elements would be computed (i.e., the first argument would be set to `x^2` and the second to `x^3`). Then, the assignment by `Set` would have catastrophic consequences.

```
x^2 = x^3
```

The following is analogous.

```
2 = 3
```

We now look at this fact in `Set` with an evaluated left-hand side.

```
x = 2

Evaluate[x] = 3
```

This sequence shows clearly, at which time the attributes become effective. `Hold` prevents the computation of its argument because of the `HoldAll` attribute it carries.

```
Hold[ReleaseHold[Hold[1 + 1]]]
```

But with `Evaluate`, we can disable an attribute like `Hold` for the arguments.

```
Hold[Evaluate[ReleaseHold[Hold[1 + 1]]]]
```

We return to `Set`. `Set` computes the arguments on the left-hand side before it carries out the assignment. Thus, the following definition for `f` is associated with `f[2]`.

```
Remove[f];
f[1 + 1] := {1, 1};
??f

f[1 + 1]
```

Here the argument cannot be further evaluated.

```
Remove[f, x, y];
f[x_ + y_] := {x, y};
??f
```

But in applying this function, the argument (in this case `1 + 2`) is first computed, and then the rules for `f` are applied. For this pattern, we do not have anything suitable defined for `f`.

```
f[1 + 2]
```

For a general argument that is the sum of two parts, the pattern fits because `x + y` cannot be further evaluated.

```
Remove[ξ, η];
f[ξ + η]
```

Because of the `Flat` attribute of `Plus`, sums with more than two terms are also matched by the definition of `f`.

```
f[ξ + η + ω + τ]
```

If we give `f` a `Hold` attribute, the example `f[1 + 2]` also works.

```
Remove[f];
SetAttributes[f, HoldFirst];
f[x_ + y_] := {x, y}

f[1 + 2]
```

The following behavior also comes up frequently. A recursive definition of a symbol does not lead to a recursive application of the definition.

```
my$RecursionLimit = $RecursionLimit;
Clear[x];
$RecursionLimit = 20;
x := x;
x
```

However, if we carry out an additional (in this case, trivial) operation on the right-hand side, we then get into an infinite loop.

```
Clear[x];
$RecursionLimit = 20;
x := CompoundExpression[x];
x
```

The difference between the two inputs can best be seen in the `FullForm`.

```
(Clear[x];
 $RecursionLimit = 20;
 x := x;
 x) // Hold // FullForm

(Clear[x];
 $RecursionLimit = 20;
 x := CompoundExpression[x];
 x) // Hold // FullForm
```

In the last case, a `CompoundExpression` is in the right-hand side of the definition. Of course, the following example does not work either.

```
Clear[x];
$RecursionLimit = 20;
x := (x; );
x

$RecursionLimit = my$RecursionLimit;
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

Overview

```
Get[ToFileName[ReplacePart[
           "FileName" /. NotebookInformation[EvaluationNotebook[]],
            "ChapterOverview.m", 2]]];
```

```
ChapterOverview["Programming", 4]
```

Exercises

1.^{L1} Explain the Errors

Why do the following inputs generate messages?

a) `a + b = 5`

b) `a = 3; a[x_] = x`

c) `(3 + 5)[[1]]`

d) `f[x_] = x_`

e) `expression = TreeForm[6 + u^(Sin[r + 78 z^z])]; expression[[2]]`

f) `x = With[{x = x}, x^12]`

g) `Set[##]&[1, 2]`

h) `f[1]([1], [1])`

i) `Remove[f]; f[x_] := (f[y_] = f[y]); f[1]`

j) `Remove[f]; f[x_] := x[f[x[f]]]; Short[f[1], 12]`

k) `Remove[f]; f[x_] := (f[y_] = f[x][y]); f[1]`

l) `Length[Sin[1, 2, 3, 4]]`

m) `headOnly = a1b2c3[1][2][3]; headOnly[[2]]`

n) `(#2. + #1.)&[1, 2]`

o) `Remove[f]; f[x_] = Function[x, x^2]; f[1]`

p) `Remove[x, f1, f2]; x /: f1_[_, f2_[x], _] := f1 f2 x`

q) `Remove[p]; p = 1; p /: Hold[p] = 0; 1/Hold[p]`

r) `mySet = Set; myVar = 1; #1[#2, #3]&[mySet, myVar, 2]`

s) `Module[{Slot}, (#1^2&[3])[[1, 1]]][[2]]`

t)
```
(f1[x_] = Block[{x = x}, x^2];

 f2[x_] := Block[{x = x}, x^2];

 {f1[2], f2[2]})
```

u)
```
Function[a, Block[{a}, a], {HoldAll}] @
        (Function[𝒶, Function[a, 𝒶 + a]][x][[1]])
```

v)
```
Function[Slot[Slot[1]]][2]

Block[{ν = 1}, Slot[ν]&[Pi][[1]] - (Evaluate[Slot[ν]]&[Pi])]
```

w)
```
Module[Evaluate[{a = 1}], a^2]

Module[Unevaluated[Unevaluated[{a = 1}]], a^2]
```

2.^{L1} `Unevaluated` and `Evaluate`

a) The standard procedure for the computation of a *Mathematica* expression is altered for expressions containing an `Unevaluated`. Examine the following, and draw some conclusions.

```
Plus[Unevaluated[1], Unevaluated[2]]

plus[Unevaluated[1], Unevaluated[2]].
```

b) Explain the result of `Nest[Set[Evaluate[Unique[x]], #]&, 1, 4]`. What happens in this construction without the `Evaluate`?

3.^{L1} `Alias[]`

Using `Information`, `?`, or `??` we can get some information on *Mathematica* commands. `Alias[]` provides an overview of those *Mathematica* commands for which an abbreviation exists. Examine them.

4.^{L1} Built-in *builtInCommand* [ ]

Examine how built-in commands react to the wrong number of arguments, for example, to none at all.

5.^{L1} Explain the Problem, Puzzle

a) The following simple implementation of an alternative to the function `plus` for adding two integers has problems with `plus[m[3], m[4]]`. Use `Trace` to see what happens.

```
Remove[plus];

SetAttributes[plus, {Flat, Orderless}]

plus[m[i_], m[j_]] := plus[m[i + 1], m[j - 1]]

plus[m[i_], m[0]] = m[i];
```

b) Find an expression *expr* that has zero length (meaning `Length[`*expr*`]` gives 0), small depth (meaning `Depth[`*expr*`]` is less or equal to 2) and is big (meaning `ByteCount[`*expr*`]` is $\geq 10^6$). (Do not use any tricks like unprotecting `Length` and/or `Depth` and/or `ByteCount`.)

6.^{L1} Predictions

a) Predict the result of the following inputs.

```
globalVar = True;

f[x_Symbol, n_Integer] :=
Module[{sum = 0}, globalVar = False;
       CheckAbort[Do[sum = sum + If[globalVar, 0, x[i]],
                     {i, n}]; globalVar = True; sum,
                  Print[Length[sum]];
                  globalVar = True; Abort[]]]
```

b) Does the following input evaluate to 0?

```
Module[{x = ξ}, Function[x, x] - Function[x + 0, x] +
                Function[x, x + 0] - Function[Evaluate[x], x]]
```

c) Will the following input issue messages? If yes, what kind of messages are to be expected?

```
Block[{Message, C, Do},
      C[Sin[1, 1], 0/0, 0^0, Do[k, {k, I, 2I}]]]
```

d) Predict the results of the following two inputs.

```
Table[ξ[1][1], {ξ[1][1], 3, 4}, {ξ[1], 1, 2}]
```

```
Table[ξ[1][1], {ξ[1], 1, 2}, {ξ[1][1], 3, 4}]
```

e) Predict the result of the following inputs.

```
f[SetAttributes[f, HoldAll], 1 + 1]
```

```
CompoundExpression[SetAttributes[g, HoldAll], g][1 + 1]
```

f) Predict the result of the following input.

```
Exp[2 I Pi] - (Exp := 2)/(I := Pi)
```

g) Will the following two inputs give the same result?

```
Sum[1/((k + 1/2)^2 + 1), {k, -Infinity, Infinity}]
Sum[1/(k^2 + 1), {k, -Infinity + 1/2, Infinity + 1/2}]
```

7.^{L2} Contexts

Predict the result of the following inputs.

a)

```
BeginPackage["question1`"]
f1::usage = " ... is the question here ..."
Begin["`Private`"]
f1[x_String] := (ToExpression[x]; xAx1 + xAx2)
End[]
EndPackage[]

f1["xAx1 = 1; xAx2 = 2; "]

f1["question1`Private`xAx1 = 1;
    question1`Private`xAx2 = 2; "]
```

b)

```
BeginPackage["question2`"]
f2::usage = " ... is also the question here ..."
Begin["`Private`"]
```

```
f2[x_String] := Module[{x1 = x, x2}, ToExpression[x]; x1 + x2]
End[]
EndPackage[]

f2["x1 = 1; x2 = 2; "]
```

c)

```
BeginPackage["question3`"]
f3::usage = " ... is still the question here ..."
Begin["`Private`"]
f3[x_String] := Module[{x = x}, ToExpression[x]; x]
End[]
EndPackage[]

f3["x"]
```

d)

```
xa = 5; xb = 6;
f4[x_String] := (Begin["context4`"]; ToExpression[x];
                 Print[ToExpression["xa + xb"]]; End[]; )

f4["xa = 1; xb = 2"];

f4["context4`xa = 1; context4`xb = 2"];

f4["xa = 11; xb = 22"];
```

e)

```
A`𝕗[x_Real] := x

B`𝕗[x_Integer] := x^2

$ContextPath = {"Global`", "System`", "A`", "B`"};

𝕗[2] // N
```

8.^{L1} `2 + I` versus `Complex[2, I]`

What happens to the input of `2 + I` as compared with the input `Complex[2, 1]`?

9.^{L1} Local Values in `Block`

`Block` allows local values of variables. Which values (downvalues, ownvalues, …) are local? When attributes are set inside a `Block` for a local variable, are they local too? What will be the result of evaluating `(a = 1; Block[{a}, Remove[a]]; a)`?

10.^{L2} `Remove[`*f*`]`

What will be the result of the following inputs?

a) `(Remove[f]; f[x_] := x + 1; f[1] + f[1, 1])`

b)
```
Remove[f]
f[x_] := x + 1
f[1] + f[1, 1]
```

Solutions

1. Explain the Errors

a) The left-hand side has the head `Plus` that has the attribute `Protected`, and thus no rule (without using `Unprotect[Plus]`) can be identified with it.

```
a + b = 5
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

b) First `a` is computed to be 3. In the computation of `a[x]`, this value of `a` is substituted, leading to `3[x_]`. The head of this object is the head of the number 3 and is thus `Integer`. Because the symbol `Integer` also carries the attribute `Protected`, no rule can be associated with it.

```
a = 3; a[x_] = x
```

After unprotecting the integers, we can associate a definition with them.

```
Unprotect[Integer];

3[x_] := x^2

3[y]
```

We restore the old behavior with respect to integers.

```
3[x_] = .

Protect[Integer];
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

c) First 3+5 is calculated to be 8. This number has length 0 (no nontrivial `TreeForm`), and thus no first part can be extracted.

```
(3 + 5)[[1]]
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

d) Here, no real "error" message is generated; only a warning message results. In almost all cases, we do not want to use a function to generate a pattern.

```
f[x_] = x_
```

The definition for `f` is applied to any argument.

```
f[y]

f[4]
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

e) Indeed, `6 + u^(Sin[r + 78 z^z])` has a second part, namely, `u^(Sin[r + 78 z^z])`, but the `Tree`

`Form` of an arbitrary expression possesses only one argument, namely, the expression itself. For the expression under consideration, the `TreeForm` is as follows.

```
TreeForm[Plus[6, Power[u, Sin[Plus[r, Times[78, Power[z, z]]]]]]]
```

So, we get a `Part::partw` message.

```
expression = TreeForm[6 + u^(Sin[r + 78 z^z])];
expression[[2]]
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

f) We first look at the result.

```
x = With[{x = x}, x^12]
```

Here is the calculation of the right-hand side alone.

```
Remove[x]
With[{x = x}, x^12]
```

After the `With` is computed, the result is assigned to `x`, and then the `x` in `x^12` is calculated with this definition. This happens often.

```
Log[12, %%%[[2]]]
```

The pure statement `x = x^12` would give a similar result.

```
x = x^12
```

Using a function that does not evaluate its arguments, we can avoid the above recursion.

```
x := With[{x = Hold[x]}, Hold[x]]

x
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

g) The input generates an error message.

```
Set[##]&[1, 2]
```

After evaluation of the pure function, `Set[##]&[1, 2]` leads to `Set[1, 2]` (which is just `1 = 2`), which then generates the error `message Set::setraw`, because the integer 1 cannot be assigned the value 2.

Also, unprotecting integers does not allow us to make assignments to the ownvalues of raw types.

```
Unprotect[Integer]

1 = 2
```

But `DownValues` and `SubValues` can now be associated with integers (identifying `1` with `Integer[1]`).

```
1[2] = 3;
SubValues[Integer]

1[2][3] = 4;
SubValues[Integer]
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

h) Arguments must always be enclosed in square brackets, so the following is not allowed syntax in *Mathematica*.

```
f[1]([1], [1])
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

i) This recursive function definition clearly leads to an infinite loop.

```
f[x_] := (f[y_] = f[y]); f[1]
```

Here is the current definition of `f`.

```
??f
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

j) This definition is also obviously recursive. To avoid writing out the long result, we apply `Short`.

```
f[x_] := x[f[x[f]]]; Short[f[1], 4]
```

Indeed, the result consists of nearly 100000 characters.

```
Characters[ToString[%]] // Length
```

According to the standard setting of `$RecursionLimit`, the above function was iterated about 256 times.

```
Depth[%%]
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

k) Here is still one more recursive definition of this type. Because of the use of a named pattern variable on the right-hand side, the variable `y$` appears here. To avoid a long output, we apply `Short`.

```
f[x_] := (f[y_] = f[x][y]); Short[f[1], 4]
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

l) *Mathematica* tries to find `Sin[1, 2, 3, 4]`. However, because the built-in function `Sin` expects one argument, we get the "error" message `Sin::argx`. The result of the computation is `Sin[1, 2, 3, 4]`. Applying `Length` to this expression gives the number of arguments, that is, 4.

```
Length[Sin[1, 2, 3, 4]]
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

m) `a1b2c3[1][2][3]` has no second part.

```
headOnly = a1b2c3[1][2][3]; headOnly[[2]]

TreeForm[headOnly]
```

It has only a first part, namely, 3. The head is `a1b2c3[1][2]`.

```
Head[headOnly]
```

We get the 2 in `headOnly` as follows.

```
headOnly[[0, 1]]
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

n) This " expression" is not syntactically correct.

```
(#2. + #1.)&[1, 2]
```

It is correct without the decimal points.

```
(#2 + #1)&[1, 2]
```

And it is also correct with a digit after the points.

```
(#2.0 + #1.0)&[1, 2]
```

But note the `FullForm` in this case.

```
#2.0 + #1.0& // FullForm
```

```
#2.0 + #1.0&[1, 2]
```

Because the argument of `Slot` must be a nonnegative integer, no short inputform exists for other arguments.

```
Slot[1.0] // InputForm
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

o) The function definition associated with `f` is `Function[x, x^2]`. If `f` is called with an argument *arg*, every `x` in `Function[x, x^2]` is replaced by *arg*. Thus, we get for `f[1]` the result `Function[1, 1^2]`. However, 1 is not allowed as a variable in the first argument of `Function`, and so the error message `Function::flpar` is generated.

```
f[x_] = Function[x, x^2]
f[1]
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

p) The `x` with which the definition is to be associated is too deeply nested to make the association.

```
x /: f1_[_, f2_[x], _] := f1 f2 x
```

This can be seen in the `TreeForm` of the left-hand side of the function definition.

```
f1_[_, f2_[x], _] // TreeForm
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

q) Here is what happens.

```
p = 1;
p /: Hold[p] = 0;
1/Hold[p]
```

The standard evaluation procedure is going on, and the `Hold` causes the `p` inside `Hold[p]` not to be evaluated. Then, the upvalues for `p` are tested and the upvalue for `Hold[p]` is used, with the result 1/0, which yields the message `Power::infy`.

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

r) The assignment `myVar = 2` cannot be done this way.

```
mySet = Set; myVar = 1;
#1[#2, #3]&[mySet, myVar, 2]
```

The reason is that first all arguments of the pure function `#1[#2, #3]&` are evaluated, which yields the three values `Set, 1, 2`, and then `Set[1, 2]` results in the error message `Set::setraw`. The same problem occurs here.

```
mySet[1, 2]
```

Using a pure function with an attribute, we can avoid the problem of evaluation.

```
mySet = Set; myVar = 1;
Function[{x1, x2, x3}, x1[x2, x3], {HoldAll}][mySet, myVar, 2]
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

s) The `Module` creates a local variable for `Slot`. This local variable does not act "properly" inside `Function`. As a result, we get `Slot$` *number*`[1]`^{2}.

```
Module[{Slot}, (#1^2&[3])]
```

Extracting two times, the first part inside the `Module` yields just 1.

```
Module[{Slot}, (#1^2&[3])[[1, 1]]]
```

`1` is an atom, and one cannot extract its second element. So, we end up with a `Part::partd` message.

```
Module[{Slot}, (#1^2&[3])[[1, 1]]][[2]]
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

t) The assignment with `Set` works fine. It is the assignment using `SetDelayed` that later on generates the message when `f2[2]` is evaluated. In evaluating the variable initialization, the expression `2=2` is encountered because 2 gets substituted for each of the three occurrences of `x` on the right-hand side of `f2`.

```
(f1[x_]  = Block[{x = x}, x^2];
 f2[x_] := Block[{x = x}, x^2];
 {f1[2], f2[2]})
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

u) The argument of the outer pure function is `Function[𝒶, Function[a, 𝒶 + a]][x][[1]]`. If evaluated, this expression gives `a$`. But the `HoldAll` attribute of the outer pure function avoids this evaluation and sticks the unevaluated expression into `Block[{a}, a]` for `a`. Because of the `HoldAll` attribute of `Block`, there it also does not get evaluated. But the elements of the first argument of `Block` must be symbols. So, we get the `Block::lvsym` message.

```
Function[a, Block[{a}, a], {HoldAll}] @
          (Function[𝒶, Function[a, 𝒶 + a]][x][[1]])
```

Using an outer pure function without the `HoldAll` attribute gives `a$`.

```
Function[a, Block[{a}, a], {}] @
          (Function[𝒶, Function[a, 𝒶 + a]][x][[1]])
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

v) Here, we evaluate the first input.

```
Function[Slot[Slot[1]]][2]
```

The first input gives `Function::slot` messages and evaluates to `0`. `Slot[ν]&[Pi]` gives the message because ν is not a nonnegative integer. But nevertheless, the `Part` command then extracts the `Pi` from the unchanged `Slot[ν]&[Pi]` expression.

```
Block[{ν = 1}, Slot[ν]&[Pi][[1]] - (Evaluate[Slot[ν]]&[Pi])]
```

The second input will not evaluate nontrivially. The reason is the low precedence of `&`. The body of `Block` is parsed as `((Slot[ν]&)[π][[1]]- Evaluate[Slot[ν]]&)[π]`. The body of the last pure function does not contain and valid `Slot`-object and so stays unchanged.

```
Block[{ν = 1}, Slot[ν]&[Pi][[1]] - Evaluate[Slot[ν]]&[Pi]]

FullForm[%]
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

w) The first input gives an error message because the `Evaluate` forces the first argument of `Module` to evaluate to `1` which is not a symbol.

```
Module[Evaluate[{a = 1}], a^2]
```

The second input the outer `Unevaluated` is stripped out and the resulting expression `Unevaluated[{a = 1}]` does not have the head `List` as required for the first argument of `Module`.

```
Module[Unevaluated[Unevaluated[{a = 1}]], a^2]
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

2. Unevaluated and Evaluate

a) We first look at what happens. The following input evaluates to 3.

```
Plus[Unevaluated[1], Unevaluated[2]]
```

```
plus[Unevaluated[1], Unevaluated[2]]
```

Now, we make our definition of `plus`.

```
plus[a_, b_] = pluplu[a, b]
```

This input leads to a different result.

```
plus[Unevaluated[1], Unevaluated[2]]
```

Here is what happened: We find that in `f[Unevaluated[x], … ]` *Mathematica* removes `Unevaluated`, while retaining a copy of the original expression. Now, if *Mathematica* finds an applicable rule (in this case, for `Plus`), it is applied, and the result is output. If *Mathematica* cannot find a rule, the original expression is returned. We look at this process again in detail.

```
Clear[plus];
On[];
plus[Unevaluated[1], Unevaluated[2]]
```

```
Off[];
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

b) Let us first look at the immediate result.

```
Nest[Set[Evaluate[Unique[x]], #]&, 1, 4]
```

But we have a side effect.

```
Names["x*"]
```

```
Function[var, Definition[var], {Listable}][%]
```

Now let us explain what is happening. The first step after the arguments in `Nest[`*func*`,` *start*`,` *iter*`]` have been evaluated is the calculation of *func*`[`*start*`]`, which in this case is `Set[Evaluate[Unique[x]], #]&[1]` or rewritten `Evaluate[Unique[x]] = 1`. The `Evaluate` of the left-hand side creates a unique variable beginning with lowercase `x`. Then, the value 1 is attached to this variable. The result of this assignment is the value 1. Then, `Nest` again takes the function from its first argument and calls it with the result from the first function evaluation. This again is a new variable, and it gets the value 1, and so on.

Without the `Evaluate` command, the above construction would not work. Because of its `HoldFirst` attribute, `Set` does not evaluate its first argument, which means no variables `x$`*number* are ever created in this situation. As a result, `Set` tries to associate the value 1 with `Unique[x]` and not with `x$`*number*. But the head of `Unique[x]`, that is, `Unique`, has the attribute `Protected` and nothing can be associated with it. That is why in this case we only get three error messages, and no assignments occur.

```
Remove["x*"]
```

```
Nest[Set[Unique[x], #]&, 1, 4]
```

```
Names["x*"]

Function[var, Definition[var], {Listable}][%]
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

3. Alias[]

Here, the input is executed.

```
Alias[]
```

For better readability, we format the result as a table.

```
Module[{l = List @@ Alias[], λ}, λ = Length[l];
TableForm[{Take[l, {1, Ceiling[λ/2]}],
          (* equal column length *)
          If[OddQ[λ], Append[#, " "], #]&[Take[l, {Ceiling[λ/2] + 1, λ}]]},
          TableDirections -> {Row, Column}, TableSpacing -> {2, 0.1}]]
```

We are already familiar with some of these short forms; most of the others will be discussed. Note that no command `SetAlias` exists. Its obvious effect can be partially realized with `$Pre` and `MakeExpression`; we do not go into this in detail here because in the current version of *Mathematica*, it is not possible (without writing a new parser) to introduce arbitrary shortcuts from the user.

When using `StandardForm` for inputting expressions, the reader can add many more rules to interpret arbitrary structures. (But because of the predefined grouping and precedence rules for operators, the spacings might not be the desired ones.)

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

4. Built-in *builtInCommand* [ ]

We cannot answer this question completely here because too many possibilities exist for calling a function with an incorrect number of arguments. As an example, we look at the case where there is no argument: *builtInCommand*`[]`. The following program would provide an overview of the problem (if it were to be executed, which we do not do here because it generates too many messages). We use `count` to count how many built-in commands generate an error message when called without an argument; those commands that produce a nontrivial result are collected in the list `bag`. (The details of the programming of the following code will only become clear later.) We do not let the program run because it generates hundreds of error messages. `systemCommands` is a list of the names of all *Mathematica* commands visible in a fresh *Mathematica* session.

We now remove some of the commands in `systemCommands`; they would either quit the *Mathematica* session or cause the program to hang.

```
systemCommands = Names["System`*"];

systemCommands =
DeleteCases[systemCommands,
            (* remove dangerous functions *)
            "Abort" | "Break" | "Continue" | "Dialog" | "Exit" | "Quit" |
            "ExitDialog" | "Edit" | "EditDefinition" | "EditIn" |
            "System`Dump`EditString" | "Goto" | "Throw" | "On[]" |
            "System`Convert`HTMLDump`BlankGIFFile" | "TraceDialog" |
            "FileBrowse" | "Experimental`FileBrowse" | "NotebookRead" |
            "Experimental`FindTimesCrossoverDigits" | "ConsoleMessage" |
            "Print" | "Internal`FromDistributedTermsList" |
            "System`Private`GetInputHeld" | "Input" | "InputString"|
            "NotebookCreate" | "Interrupt" | "FrontEnd`NotebookPut"|
            "NotebookOpen" | "NotebookPut" | "FrontEnd`PageCellTags" |
            "$Inspector" | "FrontEnd`DoHTMLSave" | "FrontEnd`DoTeXSave"];
```

Here is the actual “program”.

```
(* initialize counter and bag *)
bag = {};
count = 0;
(* test all  functions from systemCommands *)
Do[temp = systemCommands[[i]];
   check = Check[expr = ToExpression[StringJoin[temp, "[]"]], "Error"];
  (* put in bag *)
  If[check == "Error", count = count + 1,
     If[ToString[expr] != StringJoin[temp, "[]"],
        AppendTo[bag, temp]]], {i, Length[systemCommands]}];
```

Here is the result for `count`.

```
411
```

And here is the result for `bag`.

```
{AbsoluteTime, BitAnd, BitOr, BitXor, Context, Directory,
 DiscreteDelta, GCD, HomeDirectory, InString, KroneckerDelta,
 MaxMemoryUsed, MemoryInUse, Multinomial, Out, ParentDirectory, Plus,
 Power, Random, SessionTime, Share, StringJoin, Times, TimeUsed,
 TimeZone, TraceLevel, UnitStep}
```

Here, we print their values. Note that most are system and session specific.

```
Print[StringJoin[#, "[] = "], ToExpression[StringJoin[#, "[]"]]]& /@ bag
```

```
AbsoluteTime[] = dependentOnTheComputer
BitAnd[] = -1
BitOr[] = 0
BitXor[] = 0
Context[] = "Global`"
Directory[] = dependentOnTheComputer
GCD[] = 0
HomeDirectory[] = dependentOnTheComputer
InString[] = bag
```

```
MaxMemoryUsed[] = about2519248
MemoryInUse[] = about2343324
Multinomial[] = 1
Out[] = dependentOnTheInputHistory
ParentDirectory[] = dependentOnTheComputer
Plus[] = 0
Power[] = 1
Random[] = dependentOnTheComputer
SessionTime[] = dependentOnTheComputer
Times[] = 1
TimeUsed[] = dependentOnTheComputer
TimeZone[] = dependentOnTheComputer
TraceLevel[] = 0
UnitStep[] = 1
```

5. Explain the Problem, Puzzle

a) Here is the definition for `plus`.

```
SetAttributes[plus, {Flat, Orderless}]
plus[m[i_], m[j_]] := plus[m[i + 1], m[j - 1]]
plus[m[i_], m[0]] = m[i];
```

To reduce execution time, we reduce `$IterationLimit`.

```
$IterationLimit = 20
```

Here is the shortened result of `Trace[plus[m[3], m[4]]]`.

```
plus[m[3], m[4]] // Trace // Short[#, 20]&
```

The problems arise from the `Orderless` attribute. By the definition above, `plus[m[3], m[4]]` is computed to be `plus[m[4], m[3]]`. However, because of the `Orderless` attribute, this intermediate result is changed to `plus[m[3], m[4]]`, which is the starting point, and so on.

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

b) The two functions `Length` and `Depth` care about the arguments of an expression. They do not analyze the structure of the head. This means that using a large expression as the head and using zero arguments is a natural solution of the problem. Here is an explicit example.

```
expr = Nest[C, C, 10^5][];
{Length[expr], Depth[expr], ByteCount[expr]}
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

6. Predictions

a) We run the code under consideration.

```
globalVar = True;

f[x_Symbol, n_Integer] :=
Module[{sum = 0}, globalVar = False;
       CheckAbort[Do[sum = sum + If[globalVar, 0, x[i]],
                     {i, n}]; globalVar = True; sum,
                  Print[Length[sum]];
                  globalVar = True; Abort[]]]
```

The 100 kB are surely not enough to add (on computers that cannot form superpositions) about 2^{31} `x[i]`. As a result, the `MemoryConstrained` will induce an abort inside the `Do` loop. The `CheckAbort` will catch this abort and evaluate the second argument of `CheckAbort`. This evaluation resets the value of `globalVar` to `True`, and the result of the next input `globalVar` is `True`.

```
MemoryConstrained[f[x,
         Developer`$MaxMachineInteger - 1], 10^5];

globalVar
```

In the next input, the 100 kB memory limit is surely enough to add the 10 `x[`*i*`]`. No abort gets generated in this case.

```
MemoryConstrained[f[x, 10^1], 10^5]

globalVar
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

b) No, the expression does not evaluate to 0. Actually, all four terms are different.

```
Module[{x = ξ},
{Function[x, x], Function[x + 0, x],
 Function[x, x + 0], Function[Evaluate[x], x]}]
```

The first expression `Function` just stays as it is. The first argument of the second function is not a symbol, so *Mathematica* does not know which symbol to keep local to `Function`. As a result, the `x$`*number* variable from `Module` slips in. But because of the `HoldAll` attribute of `Function`, these `x$`*number* do not evaluate to ξ. The third `Function` is similar to the first one. But again, because of the `HoldAll` attribute, `x + 0` does not evaluate to 0. The last `Function` again does not have a symbol as its first argument. But this time the `Evaluate` forces the `x$`*number* to evaluate to ξ.

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

c) If not embedded in other constructs and if messages are not shut off, all of the four arguments of `C` will issue messages.

```
C[Sin[1, 1], 0/0, 0^0, Do[k, {k, I, 2I}]]
```

A message is issued when `Message[MessageName[`*symbol*`, "`*tag*`"]]` is evaluated. If `Message` is a variable local to `Block`, the built-in rules for the symbol `Message` are temporarily disabled and no messages will be printed. But in addition, `Do` is a local variable in the `Block`. Inside the `Block`, `Do` will not generate a `Do::"iterb"` message call at all. But after the evaluation of `Block` the result `C[Sin[1,1],Indeterminate,Indeterminate,Do[k,{k,i,2 i}]]` is re-evaluated and now the built-in rules for `Do` generate a `Do::"iterb"` message call. The following shows this.

```
Block[{Message, C, Do, res},
      (Print["Evaluation of Block finished"]; #)&[
         C[Sin[1, 1], 0/0, 0^0, Do[k, {k, I, 2I}]]]]
```

This is exactly the message we obtain from directly evaluating the input under consideration.

```
Block[{Message, C, Do},
      C[Sin[1, 1], 0/0, 0^0, Do[k, {k, I, 2I}]]]
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

d) In the first input, the outer iterator `{ξ[1][1], 3, 4}` localizes the body. Because of the `Block`-like nature of the variable localization in `Table`, the names of the variables do not change. Then the inner iterator `{ξ[1], 1, 2}` localizes the `ξ[1]` in (the already localized) `ξ[1][1]`. Consequently, the outer iterator simply causes the inner iterator to be carried out twice. The inner iterator variable takes on the values `1` and `2` and the body of the `Table` evaluates to `1[1]` and `2[1]`. As a result, the first input returns `{{1[1],2[1]},{1[1],2[1]}}`.

```
Table[ξ[1][1], {ξ[1][1], 3, 4}, {ξ[1], 1, 2}]
```

In the second input, the outer iterator `{ξ[1], 1, 2}` localizes the `ξ[1]`. `ξ[1]` appears in the body of `Table` as well in the inner iterator. In carrying out the inner iterator `{ξ[1][1], 3, 4}` for localized `ξ[1]` assignments of the form `1[1] = 3`, `1[1] = 4`, `2[1] = 3`, and `2[1] = 4` are created. These assignments lead to `Set::write` messages. Because these assignments fail, the inner iterator only causes the creation of a list with two identical elements. The elements themselves are solely determined by the outer iterator. The first value of the outer iterator produces `1[1]` and the second `2[1]`. As the result, the list `{{1[1],1[1]},{2[1],2[1]}}` is returned.

```
Table[ξ[1][1], {ξ[1], 1, 2}, {ξ[1][1], 3, 4}]
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

e) At the time the head `f` gets evaluated `f` does not have an attribute. According to the general evaluation order, the arguments are evaluated next. The first argument adds the `HoldAll` attribute to `f`. Then immediately the second argument gets evaluated. So the result is `f[Null,2]`.

```
f[SetAttributes[f, HoldAll], 1 + 1]
```

If inside the second argument there would be again a function `f`, the `HoldAll` attribute would go into effect.

```
Remove[f];
f[SetAttributes[f, HoldAll], f[1 + 1]]
```

In the second input, the head of `(SetAttributes[g,HoldAll];g)[1+1]` is evaluated first. This sets the `HoldAll` attribute for `g`. Consequently, the argument 1 + 1 will not be evaluated and the result is `g[1+1]`.

```
CompoundExpression[SetAttributes[g, HoldAll], g][1 + 1]
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

f) The result is 0. `Exp[2I Pi]` evaluates to 1. The (attempted) two `SetDelayed` assignments to the protected symbols `Exp` and `I` both fail, generate warning messages, and evaluate to `$Failed`. The ratio `$Failed/$Failed` evaluates to `1` and the difference evaluates to `0`.

```
Exp[2 I Pi]  - (Exp := 2)/(I := Pi)
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

g) We start with the sum $\sum_{k=-\infty}^{\infty} 1 \big/ \left(k^2+1\right)$. Its value is $\pi \coth(\pi)$. With the lower bound `-Infinity`, *Mathematica* effectively calculates the sum starting from the integer 0 to $-\infty$ in steps of -1.

```
Sum[1/(k^2 + 1), {k, -Infinity, Infinity}]
```

Shifting k to $k + 1/2$ in each summand gives a sum with value $\pi \tanh(\pi)$.

```
Sum[1/((k + 1/2)^2 + 1), {k, -Infinity, Infinity}]
```

But shifting the iterator limits by $1/2$ gives again $\pi \coth(\pi)$.

```
Sum[1/(k^2 + 1), {k, -Infinity + 1/2, Infinity + 1/2}]
```

The last result can be understood by taking into account the evaluation order of *Mathematica* expressions and the fact that infinite quantities (here `DirectedInfinity[-1]` and `DirectedInfinity[1]`) “absorb” any finite (real or complex) quantity. So, before the actual summation process happens, the iterator evaluates to `{k, -Infinity, Infinity}`.

```
{-Infinity + 1/2, Infinity + 1/2}
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

7. Contexts

a) Here is the evaluation of the first two inputs.

```
BeginPackage["question1`"]
f1::usage = " ... is the question here..."
Begin["`Private`"]
f1[x_String] := (ToExpression[x]; xAx1 + xAx2)
End[]
EndPackage[]
```

```
f1["xAx1 = 1; xAx2 = 2; "]
```

At the time of evaluation of `f1["xAx1 = 1; xAx2 = 2; "]`, the context was `Global`` (at the time of making the definition for `f1`, it was `question1a`Private``). This can be clearly seen if we write out the current context during the computation. Here is a copy of the inputs from above (we rename the function `f1` to `f1a`).

```
BeginPackage["question1a`"]
f1a::usage = " ...  is the question here ..."
Begin["`Private`"]
CellPrint[Cell[TextData[{"∘ The current context is ",
               StyleBox[Context[], "MR"], "."}], "PrintText"]];
f1a[x_String] := (ToExpression[x];
   CellPrint[Cell[TextData[{"∘ Now, the context is ",
                           StyleBox[Context[], "MR"], "."}],
                  "PrintText"]];
                 xAx1 + xAx2)
End[]
EndPackage[]
```

```
f1a["xAx1 = 1; xAx2 = 2; "]
```

However, the sum is formed with `xAx1` and `xAx2` from the context `question1`Private``. This can be seen by looking at the definition of `f1`.

```
??f1
```

These variables have not yet been assigned any values.

```
Names["*`xAx*"]
```

If we assign explicit values to these variables, we get a numerical result.

```
f1["question1`Private`xAx1 = 1;
    question1`Private`xAx2 = 2; "]
```

Now, of course, `f1["xAx1 = 1; xAx2 = 2; "]` also evaluates to 3.

```
f1["xAx1 = 1; xAx2 = 2; "]
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

b) We again look at the result of these two inputs.

```
BeginPackage["question2`"]
f2::usage = " ... is also the question here ..."
Begin["`Private`"]
f2[x_String] := Module[{x1 = x, x2}, ToExpression[x]; x1 + x2]
End[]
EndPackage[]
```

```
f2["x1 = 1; x2 = 2; "] // FullForm
```

The local `x1$`*n* from the context `question2`Private`` is assigned the value of the string `"x1 = 1; x2 = 2; "`. The local `x2$`*n* in the context `question2`Private`` remains uncomputed, because no value was assigned to it at the beginning of `Module`. At the time of the evaluation of the `Module`, the context will be `Global``.

By adding another `CellPrint`, we see the context of the local version of the variable `x1`.

```
BeginPackage["question2`"]
f2a::usage = " ... is also the question here ..."
Begin["`Private`"]
f2a[x_String] := Module[{x1 = x, x2},
  CellPrint[Cell[TextData[{"∘ The current context is ",
                StyleBox[Context[], "MR"], "."}], "PrintText"]];
                    ToExpression[x]; x1 + x2]
End[]
EndPackage[]
```

```
f2a["x1 = 1; x2 = 2; "]
```

`ToExpression` creates the symbol, but the result remains unused.

```
??x1
```

`x2` never got assigned a value.

```
??x2
```

Using `Block` instead of `Module` gives in a similar result. This time, no `x2$`*number* is created.

```
BeginPackage["question2`"]
f2b::usage = " ... is also the question here ..."
Begin["`Private`"]
f2b[x_String] := Block[{x1 = x, x2},
  CellPrint[Cell[TextData[{"∘ The current context is ",
                StyleBox[Context[], "MR"], "."}], "PrintText"]];
                    ToExpression[x]; x1 + x2]
End[]
EndPackage[]
```

```
f2b["x1 = 1; x2 = 2; "]
```

Inside `Module`, a `ToExpression` call will generate a variable without automatically appending a `$`*number*.

```
Module[{x3 = 2}, ToExpression["x3 = 3"]; x3]
```

Inside `Block`, on the other hand, a `ToExpression` can easily influence the value of a variable.

```
Block[{x3 = 2}, ToExpression["x3 = 3"]; x3]
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

c) In this example, problems occur with the double use of variables.

```
BeginPackage["question3`"]
f3::usage = " ...  is still the question here ..."
Begin["`Private`"]
f3[x_String] := Module[{x = x}, ToExpression[x]; x]
End[]
EndPackage[]
```

```
f3["x"]
```

We look at the `FullForm` of the results to better identify the strings.

```
FullForm[%]
```

The problems with the assignment of the local variables are not due to context issues, but stem from the double use of the variables in the left-hand side of the function definition and in `Module`.

```
generateLocalAssignmentProblem[x_] := Module[{x = x}, x^2];
generateLocalAssignmentProblem["x"] // InputForm
```

The `x` on the left in the local variables of `Module` causes the problems.

```
generateLocalAssignmentProblem[x_] := Module[{x = y}, y^2];
generateLocalAssignmentProblem["x"] // InputForm
```

This error message stems from the replacement of all `x` on the right-hand side of the function definition of `generate⋱ LocalAssignmentProblem` corresponding to the `DownValues` associated with `generateLocalAssignment Problem`.

```
DownValues[generateLocalAssignmentProblem]
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

d) In `f4["xa = 1; xb = 2"]`, the “string-values” of the arguments `xa` and `xb` are used to compute the sum.

```
xa = 5; xb = 6;

f4[x_String] :=
(Begin["context4`"]; ToExpression[x];
 Print[ToExpression["xa + xb"]]; End[]; )

f4["xa = 1; xb = 2"]
```

Here is what happens: During the evaluation of `f4`, the current context is changed. We can see this here.

```
f4a[x_String] :=
(Begin["context4`"]; ToExpression[x];
 CellPrint[Cell[TextData[{"∘ The current context is ",
                   StyleBox[Context[], "MR"], "."}], "PrintText"]];
 Print[ToExpression["xa + xb"]]; End[]; )

f4a["xa = 1; xb = 2"]
```

In evaluating `f4["xa = 1; xb = 2"]`, the symbols `xa` and `xb` appear. They do not exist in the context `context4``.

```
Names["*`xa"]
```

They are not created immediately, however, but only after it is verified whether symbols with the same names in some context of `$ContextPath` already exist, which includes the context `Global``.

```
(Begin["context5`"];
CellPrint[Cell[TextData[{"∘ The current context path is: ",
                StyleBox[ToString[InputForm[$ContextPath]], "MR"]}],
              "PrintText"]];
 End[]);
```

This is the case here, because the symbols `xa` and `xb` are present in the context `Global``. Thus, their values will consequently be changed.

```
{xa, xb, Global`xa, Global`xb}
```

So, we get the result 3. Now, consider the following example.

```
f4["context4`xa = 11; context4`xb = 22"];
```

In the evaluation of `f4["context4`xa = 1; context4`xb = 2"]`, the symbols `context4`xa` and `context4`xb` are generated in the current context `context4``. However, the context does not have to be explicitly written in the current context. Thus, in the following call on `xa` and `xb` from the current context, we use `xa(=context4`xa)` and `xb(=context4`xb)`. *Mathematica* first looks in the current context; if the symbols do not appear there, we search through the contexts in `$ContextPath`. Currently, we have the following `xa`s.

```
Names["*`xa"]
```

Here are the `xa` values.

```
xa
```

```
context4`xa
```

And here are the `xb` values.

```
xb
```

```
context4`xb
```

Here again, `context4`xa` and `context4`xb` are used, whose values are not changed.

```
f4["nothingButJustxaAndxb"]
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

e) The result is `2.`. After making the definition and changing the context path, `𝕗[2]` is evaluated. The context `A`` contains the symbol 𝕗, so *Mathematica* tries to use the definitions from this context. But for the argument 2, none of them matches. So it returns `𝕗[2]`. The definitions for 𝕗 from the context `B`` are not tried. Numericalization of the 2 (with `N`) yields an argument so that the definition for 𝕗 from the context `A`` matches and `𝕗[2.]` evaluates to the real number `2.`.

```
A`𝕗[x_Real] := x
B`𝕗[x_Integer] := x^2
$ContextPath = {"Global`", "System`", "A`", "B`"};
𝕗[2] // N
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

8. 2 + I versus Complex[2, I]

`2 + I` leads to the addition of the integer `2` and the complex number `I` (which evaluates to `Complex[0, 1]`), and the result is the complex number `Complex[2, 1]`.

```
On[]; 2 + I; Off[]
```

Here, we compare the unevaluated with the evaluated form of `I`. (Because I is a symbol and not a number, it was discussed in the Subsection 2.2.4 about constants and not in Subsection 2.2.1 about numbers.)

```
Head[Unevaluated[I]]
```

```
Head[I]
```

In contrast, for the input `Complex[2, 1]`, nothing is computed; it is already in the form of a raw object.

```
On[]; Complex[2, 1]; Off[]
```

We see the difference between the two forms `2 + I` and `Complex[2, 1]` clearly in the following.

```
{FullForm[Hold[2 + I]], FullForm[Hold[Complex[2, 1]]]}
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

9. Local Values in Block

Here is a `Block` construct. For the local variables `fo`, `fd`, `fu`, `fn`, `fs`, and `ff` we set all possible values. (This means we give an ownvalue, upvalue, a downvalue, a formatvalue, a subvalue and a numeric value).

```
Block[{fo, fd, fu, fn, fs, ff},
      fo = 1; fd[x_] := x; fu /: ℱ[fu] := 2;
      N[fn] = 1.; fs[1][y_] := y^2;
      Format[ff[z_]] := Subscript[ff, z];
      {OwnValues[fo], DownValues[fd], UpValues[fu],
       NValues[fn], SubValues[fs], FormatValues[ff]}]
```

Outside the `Block`, none of the values exists anymore.

```
{OwnValues[fo], DownValues[fd], UpValues[fu],
 NValues[fn], SubValues[fs], FormatValues[ff]}
```

Also, attributes are kept local.

```
Block[{fa}, SetAttributes[fa, Listable]; fa[{1, 2}]]

fa[{1, 2}]

??fa

Block[{fa1}, SetAttributes[fa1, Protected]]

??fa1
```

Only the attribute `Locked` can “escape”. This is to be expected. A locked variable cannot be modified anymore. So *Mathematica*’s attempts to clear the attribute when leaving the `Block` must fail.

```
Block[{fa2}, SetAttributes[fa2, Locked]]

??fa2
```

Now let us evaluate `(a = 1; Block[{a}, Remove[a]]; a)`. The result will be `Removed[a]`. Because of the parentheses, `(a = 1; Block[{a}, Remove[a]]; a)` is parsed as one expression. When evaluating this expression the symbol `a` will be removed inside the `Block`. After the removal, `a` is again used. But at this time, it is a removed variable and `Removed[a]` will be returned.

Messages are bound to variables. They do not represent values of variables. So the message associated with `fa3` in the following `Block` is available outside of `Block`.

```
Block[{fa3}, fa3::aMessage = "fa3 lives in a Block"];

fa3::amessage
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

10. Remove[*f*]

Let us first look at the results of the two inputs.

```
(Remove[f]; f[x_] := x + 1; f[1] + f[1, 1])

Remove[f]
f[x_] := x + 1
f[1] + f[1, 1]
```

The result of the second example is probably the expected one. To understand the result of the first example, we look at its `FullForm`.

```
FullForm[Hold[(Remove[f]; f[x_] := x + 1; f[1] + f[1, 1])]]
```

The head of the expression is `CompoundExpression`, so in distinction to the second example, this is one *Mathematica* expression. To see how this expression is evaluated in more detail, we use `On[]`.

```
On[]

(Remove[f]; f[x_] := x + 1; f[1] + f[1, 1])
```

Here we see what is going on: The `Remove[f]` removes the `f`. Because `f` is still needed in the other pieces of the `CompoundExpression`, the result of removing `f` is `Removed[f]`. Then, the `Set` statement `f[x_] := x + 1` is carried out. But the definition is not stored as a definition of `f`, but rather as a definition for `Removed[f]`. We can see this more clearly if we change the above code slightly.

```
Off[]

(Remove[f]; f[x_] := x + 1; DownValues[f])
```

Finally, the definition for `Removed["f"][x_]]` is used to calculate the value `2` for `f[1]`. No definition matches `f[1,1]`. As a result, we obtain `2 + Removed["f"][1,1]`.

The symbol `Removed` cannot be removed.

```
r = Removed;
Unprotect[Removed]; Remove[Removed]
r // InputForm
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

References

✱1 W. Ackermann. *Math. Ann.* 99, 118 (1928).

✱2 C. Calude, S. Marcus, I. Tevy. *Hist. Math.* 6, 380 (1974). *DOI-Link*

✱3 G. J. Chaitin. *The Unknowable*, Springer-Verlag, New York, 1998. *BookLink*

✱4 G. J. Chaitin. *The Limits of Mathematics*, Springer-Verlag, New York, 1999. *BookLink (2)*

✱5 G. J. Chaitin. *arXiv:chao-dyn*/9909011 (1999). *Get Preprint*

✱6 S. B. Cooper. *Computability Theory*, Chapman & Hall, Boca Raton, 2004. *BookLink (2)*

✱7 D. Deutsch, A. Ekert, R. Lupacchini. *Bull. Symb. Logic* 6, 265 (2000).

✱8 J. Dieudonne. *Geschichte der Mathematik*, Verlag der Wissenschaften, Berlin, 1985.

✱9 E. Fredkin in *Workshop on Physics and Computation PhysComp '92*, IEEE Computer Society Press, Los Alamitos, 1993. *BookLink*

✱10 R. P. Grimaldi. *Discrete and Combinatorical Mathematics*, Addison-Wesley, Reading, 1994. *BookLink*

✱11 J. W. Grossman, R.S. Zeitman. *Theor. Comput. Sc.* 57, 327 (1988). *DOI-Link*

✶12 N. D. Jones. *Computability and Complexity from a Programming Perspective*, MIT Press, 1997. *BookLink*

✶13 N. D. Jones in S. B. Cooper, J. K. Truss (eds.). *Models and Computability*, Cambridge University Press, Cambridge, 1999. *BookLink (2)*

✶14 T. D. Kieu. *arXiv:quant-ph*/0205093 (2002). *Get Preprint*

✶15 T. D. Kieu. *Contemp. Phys.* 44, 51 (2003). *DOI-Link*

✶16 R. Maeder. *Programming in Mathematica*, Addison-Wesley, Reading, 1991. *BookLink (3)*

✶17 K. K. Nambiar. *Appl. Math. Lett.* 8, 51 (1995).

✶18 A. Oberschelp. *Rekursionstheorie*, BI, Mannheim, 1993. *BookLink*

✶19 R. Péter. *Math. Ann.* 111, 42 (1935).

✶20 R. Péter. *Rekursive Funktionen*, Budapest, 1951. *BookLink*

✶21 R. M. Robinson. *Bull. Am. Math. Soc.* 54, 987 (1948).

✶22 H. E. Rose. *Subrecursion, Functions and Hierarchies*, Clarendon Press, Oxford, 1984. *BookLink*

✶23 M. Sharir, P. K. Agarwal. *Davenport-Schinzel Sequences and their Geometric Applications*, Cambridge University Press, Cambridge, 1995. *BookLink*

✶24 C. Smorynski. *Logical Number Theory I*, Springer-Verlag, Berlin, 1991. *BookLink*

✶25 Y. Sundblad. *BIT* 11, 107 (1971).

✶26 Z. Toroczkai. *arXiv:cond-mat*/0108448 (2001). *Get Preprint*

✶27 M. Trott. *The Mathematica GuideBook for Graphics*, Springer-Verlag, New York, 2004. *BookLink*

✶28 M. Trott. *The Mathematica GuideBook for Numerics*, Springer-Verlag, New York, 2005. *BookLink*

✶29 M. Trott. *The Mathematica GuideBook for Symbolics*, Springer-Verlag, New York, 2005. *BookLink*

✶30 A. Weiermann. *Discr. Math. Theor. Comput. Sci.* 6, 133 (2003).

http://www.dmtcs.org/volumes/abstracts/dm060111.abs.html

✶31 D. Withoff. *Mathematica Internals*. Proceedings *Mathematica* Conference, Boston, 1992 (*MathSource* 0203-982). http://library.wolfram.com/infocenter/Conferences/4683

✶32 S. Wolfram. *The Mathematica Book*, Cambridge University Press and Wolfram Media, Cambridge, 1999.

BookLink

✶33 A. Zeller. *arXiv:cs.SE*/0309047 (2003). *Get Preprint*

P R O G R A M M I N G

CHAPTER 5

Restricted Patterns and Replacement Rules

5.0 Remarks

The main topics of this chapter are replacement rules and patterns. No other available programming system comes close to *Mathematica*'s ability to match patterns in arbitrary structures (expressions). The ability to select subexpressions on the basis of their form and/or contents and to manipulate them permits the construction of very elegant, short, and direct programs. However, the use of pattern matching in very large expressions may require a lot of time because of the potential combinatorial explosion of all possible pattern realizations. But a thoughtful, appropriate use of patterns allows us to write programs that are quite elegant, fast, natural, and easy to read and to maintain. We begin this chapter with a discussion of Boolean variables and functions because the determination of truth values is an important part of constructing special patterns.

```
(* no spelling warnings, set fonts for tick labels, ... *)
Get[ToFileName[ReplacePart["FileName" /.
 NotebookInformation[EvaluationNotebook[]], "Initialization.m", 2]]];
```

5.1 Boolean and Related Functions

5.1.1 Boolean Functions for Numbers

Boolean functions find the truth value for a statement. A statement can be true, false, or indeterminate.

`True`

represents the truth value true.

`False`

represents the truth value false.

Mathematica expressions can have a truth value or they may have no truth value at all, for example, when a variable `var` is not explicitly defined or the arithmetic expression 1 + 1 also does not have an obvious truth value. Here are a few examples (the meaning of < is obvious; we discuss it further in a moment).

```
{True, False, 1 < 2, symbol, 2, E < Pi}
```

Mathematica has many commands that return truth values.

Most of the commands for tests that determine the truth value of an expression end in the letter `Q` (Question); they are also called predicates. They return either `True` or `False`, but usually do not return unevaluated. (They can return unevaluated—when they are called with an inappropriate number of arguments.)

Here are the commands ending in `Q`.

```
?? *Q
```

There are about 40 such commands. Not all of them are predicates; for instance, we discuss `PartitionsQ` in Chapter 2 of the Numerics volume [139✱] and `HypergeometricPFQ` again in Chapter 3 of the Symbolics volume [140✱].

```
Length[Names["*Q"]]
```

If we count in all contexts, we find about 70 functions ending with `Q`.

```
Length[Names["*`*Q"]]
```

The truth value of a statement can be checked with `TrueQ`.

`TrueQ[`*expression*`]`

gives `True` if *expression* has the truth value true, and `False` if the expression has the truth value false or when it cannot be determined (this means it has no truth value).

Here are a few examples of the different cases.

```
Function[isItTrue, TrueQ[isItTrue], {Listable}][
    {True, False, 1 < 2, Equal, 2, E < Pi, 2 + 2 I}]
```

Here is a more complicated example. The left-hand side of the following inequality is the radical expression of the right-hand side. Because *Mathematica* uses numerical techniques to determine the truth value of the inequality, it cannot decide if the left-hand side is smaller than is the right-hand side (within the precision used to calculate numerical approximations of the left-hand side and right-hand side expressions). As a result, a message is issued (we will discuss this particular message in detail in Chapter 1 of the Numerics volume [139✱]), and the inequality is returned unevaluated.

```
Sqrt[(5 + Sqrt[5])/32] - Sqrt[3/64](Sqrt[5] - 1) < Sin[Pi/15]
```

Applying `TrueQ` to the last result, gives `False`, not because the inequality is false, but because the expression is not `True`.

```
TrueQ[%]
```

Whether an expression is a number can be determined with `NumberQ`.

`NumberQ[`*expression*`]`

gives `True` if *expression* is a number; that is, the head is `Integer`, `Real`, `Rational`, or `Complex`; otherwise, it gives `False`.

3 is a number, but π or $\sin(1)$ or $\sqrt{2}$ are not numbers. They are numeric quantities and typically have a nontrivial tree form. If an expression is a numeric quantity, it can be checked using the function `NumericQ`.

`NumericQ[`*expression*`]`

gives `True` if *expression* is a numeric quantity, that is generically after applying `N`, *expression* evaluates to a number.

Integers and their properties to be even or odd can be checked with the following commands.

`IntegerQ[`*expression*`]`

gives `True` if *expression* is a positive or negative integer or 0, that is, if it has the head `Integer`; otherwise, it gives `False`.

`EvenQ[`*expression*`]`

gives `True` if *expression* is an even integer (…, −8, −6, −4, −2, 0, 2, 4, 6, 8, …); otherwise, it gives `False`.

`OddQ[`*expression*`]`

gives `True` if *expression* is an odd integer (…, −9, −7, −5, −3, −1, 1, 3, 5, 7, 9, …); otherwise, it gives `False`.

Here is a simple example encompassing all of these possibilities. To compare several “numbers” at once, we use the attribute `Listable`.

```
Attributes[NumberQ]

SetAttributes[{NumberQ, NumericQ, IntegerQ, EvenQ, OddQ}, Listable];
```

Here are the objects to be tested.

```
testTruthValues =
{-3, -2, -1, 0, 1, 2, 3, I, 3.3, nAn, Pi, E, 3 + 6 I, 6/7,
 0.0, Sqrt[2], N[4, 20], 0``50, 1.0 - I Sqrt[2],
 HoldPattern[2], Hold[2], Unevaluated[2], HoldPattern[2],
 Infinity, Indeterminate}
```

To put the result in an easily readable form, we generate a tabular display. (We give a detailed discussion of creating and formatting tables in the next chapter.)

```
TableForm[Transpose[{NumberQ[testTruthValues],
                     NumericQ[testTruthValues],
                     IntegerQ[testTruthValues],
                     EvenQ[testTruthValues],
                     OddQ[testTruthValues]}],
          (* the table headings *)
          TableHeadings -> {testTruthValues,
             (* in bold *) StyleForm[#, FontWeight -> "Bold"]& /@
             {"NumberQ", "NumericQ", "IntegerQ", "EvenQ", "OddQ"}},
           TableSpacing -> {1, 1}]
```

An expression is `NumericQ` when it is built from numbers, constants (such as `Pi`, `E`, `GoldenRatio`, …), and functions that have the `NumericFunction` attribute. Here is an example.

```
Sin[Pi/27 + GoldenRatio^Log[EulerGamma + I/3] -
    Tan[Tan[Tan[Tan[11^11]]]]] // NumericQ
```

Be aware that the function `NumericQ` will not check if an expression represents a finite number. So an expression *expr* that is infinity or indeterminate might still give the result `True` for `NumericQ[`*expr*`]`.

```
(* (Pi - 1)^2 - (Pi^2 - 2 Pi + 1) is mathematically identical to 0 *)
Csc[(Pi - 1)^2 - (Pi^2 - 2 Pi + 1)] // NumericQ
```

```
Csc[(Pi - 1)^2 - (Pi^2 - 2 Pi + 1)] // N[#, 22]&
```

Another special property of `NumericQ` is the possibility to give this property to individual expressions. The following input generates two identical expressions αN and βN. We make αN a numeric expressions through an upvalue.

```
αN = g[Pi, Pi]; βN = g[Pi, Pi];
```

```
NumericQ[α] ^= True
```

While αN and βN are identical expressions (in the sense of `SameQ`), `NumericQ` returns different values when applied to them.

```
{αN === βN, NumericQ[α], NumericQ[β]}
```

But the two symbols `ComplexInfinity` and `Indeterminate` are not considered to be numeric quantities.

```
{NumericQ[ComplexInfinity], NumericQ[Indeterminate]}
```

Sometimes we want to restrict the domain of a function to exact numbers and sometimes to inexact numbers. The two functions `ExactNumberQ` and `InexactNumberQ` are very useful in this respect.

`ExactNumberQ[`*number*`]`
gives `True` if *number* is an exact number.

`InexactNumberQ[`*number*`]`
gives `True` if *number* is an approximative number.

In the following, `Pi` and `Sqrt[2]` are not numbers.

```
{ExactNumberQ[2], ExactNumberQ[2/9],
 ExactNumberQ[Pi], ExactNumberQ[Sqrt[2]],
 ExactNumberQ[N[3, 200]], ExactNumberQ[2 + 3.4 I]}
```

If a complex number has an exact real part and an approximative imaginary part, it counts as an inexact number.

```
{InexactNumberQ[2], InexactNumberQ[2/9],
 InexactNumberQ[Pi], InexactNumberQ[Sqrt[2]],
 InexactNumberQ[N[3, 200]], InexactNumberQ[2 + 3.4 I]}
```

This is also an inexact number.

```
InexactNumberQ[0``100]
```

`Infinity` is not a number at all.

```
{ExactNumberQ[Infinity], InexactNumberQ[Infinity]}
```

Calling `ExactNumberQ` with two arguments gives a message, and it returns unevaluated.

```
ExactNumberQ[1, 2]
```

The following input returns `False` because the unevaluated form of 1 + 2 has the head `Plus`.

```
ExactNumberQ[Unevaluated[1 + 2]]
```

And the unevaluated form of `I` has the head symbol; only after evaluation, the expression `I` becomes `Complex[0, 1]`.

```
ExactNumberQ[Unevaluated[I]]
```

A special command checks whether a number is prime.

`PrimeQ[`*expression*`]`
gives `True` if *expression* is a (positive or negative) prime number; otherwise, it gives `False`.

Here, we test the first integers and some expressions.

```
SetAttributes[PrimeQ, Listable];
PrimeQ[{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13,
        14, 15, 16, 17, 18, 19, 20, Infinity, 0.0, 3.0}]
```

The product of -1 with a positive prime number also has the truth value `True`.

```
{PrimeQ[-2], PrimeQ[-3], PrimeQ[-5]}
```

The function `PrimeQ` also has an option.

```
Options[PrimeQ]
```

Mathematical Remark: Gaussian Prime Numbers

Prime numbers that cannot be written as the product of complex numbers with integer real and imaginary parts are called Gaussian prime numbers. Not all ordinary primes are Gaussian primes, because, for example, 2 can be factored into the product of $(1+i)(1-i)$.

`PrimeQ[`*expression*`, GaussianIntegers -> True]`
gives `True` if *expression* is a Gaussian prime number; otherwise, it gives `False`.

Here is a test on the first nine integers.

```
Table[PrimeQ[k], {k, 9}]

Table[PrimeQ[k, GaussianIntegers -> True], {k, 9}]
```

Here are the factorizations of the first five prime numbers, which are not Gaussian primes.

```
{(1 + I) (1 - I), (1 + 2I) (2 + I) (-I), (2 + 3I) (3 + 2I) (-I),
 (1 + 4I) (4 + I) (-I), (2 + 5I) (5 + 2I) (-I)}
```

Note that these factorizations can, of course, be calculated with *Mathematica*. The relevant command is `FactorInte`⋱
`ger`, which we discuss in Chapter 2 of the Numerics volume [139✱]. *Mathematica* chooses a slightly different form for the factorization, for instance, $2 = -i\,(1+i)^2 = (1-i)\,(1+i)$.

```
FactorInteger[{2, 5, 13, 17, 29}, GaussianIntegers -> True]
```

Now, we discuss < and > (which were used above). For integers, rationals, and real numbers, we can define a partial order relation with <, ≤, >, and ≥ in a “natural way”.
Two remarks are in order here:
1) `Less`, `Greater`, … are not real predicates in the sense that they end with `Q`. But for numbers as arguments, they behave as predicates and return `True` or `False`. That is why they are discussed in this subsection. For symbolic (or even sometimes exact numeric) arguments they can stay unevaluated.
2) In connection to < and >, = (`Equal` or == in *Mathematica*) should also be mentioned here. Because of its extraordinary importance for representing equations, it will be discussed in detail in the next subsection. While `Less`, `Greater` and `Equal` can all stay unevaluated, in typical uses `Less` and `Greater` will more frequently evaluate nontrivial.

`Less[`*expression*$_1$`,` *expression*$_2$`, …,` *expression*$_n$`]`

or

expression$_1$ < *expression*$_2$ < ⋯ < *expression*$_n$

gives `True` if *Mathematica* can determine that *expression*$_1$ < *expression*$_2$ < … < *expression*$_n$. If it can be checked that this does not hold, it gives `False`. If neither case can be established, the entire expression remains unevaluated. If the overall expression contains variables, and neither the truth value `True` nor `False` can be determined, *Mathematica* considers it a chain of inequalities.

`LessEqual[`*expression*$_1$`,` *expression*$_2$`, …,` *expression*$_n$`]`

or

expression$_1$ <= *expression*$_2$ <= ⋯ <= *expression*$_n$

gives `True` if *Mathematica* can determine that *expression*$_1$ ≤ *expression*$_2$ ≤ … ≤ *expression*$_n$. If it can be checked that this does not hold, it gives `False`. If neither case can be established, the entire expression remains unevaluated. If the overall expression contains variables, and neither the truth value `True` nor `False` can be determined, *Mathematica* considers it as a chain of inequalities.

`Greater[`*expression*$_1$`,` *expression*$_2$`, …,` *expression*$_n$`]`

or

expression$_1$ > *expression*$_2$ > ⋯ > *expression*$_n$

gives `True` if *Mathematica* can determine that *expression*$_1$>*expression*$_2$> … > *expression*$_n$. If it can be checked that this does not hold, it gives `False`. If neither case can be established, the entire expression remains unevaluated. If the overall expression contains variables, and neither the truth value `True` nor `False` can be determined, *Mathematica* considers it as a chain of inequalities.

`GreaterEqual[`*expression*$_1$`,` *expression*$_2$`, …,` *expression*$_n$`]`

or

expression$_1$ >= *expression*$_2$ >= ⋯ >= *expression*$_n$

gives `True` if *Mathematica* can determine that *expression*$_1$ ≥ *expression*$_2$ ≥ … ≥ *expression*$_n$. If it can be checked that this does not hold, it gives `False`. If neither case can be established, the entire expression remains unevaluated. If the overall expression contains variables, and neither the truth value `True` nor `False` can be determined, *Mathematica* considers it as a chain of inequalities.

When the truth value of an inequality cannot be determined, *Mathematica* considers the inequality as an imperative statement, a condition on the variables. Inequalities are used in this sense, e.g., in `ConstrainedMax` or `ConstrainedMin` [74✱], or in the package ``Algebra`InequalitySolve`` (or in the experimental function ``Experimental`Resolve``).

Here are a few simple examples.

```
1 < 2 < 3 < 4 < 5

2 > 1 > -6 > -9.89 > -56782/675

-Infinity < Infinity
```

```
Infinity <= Infinity
```

```
DirectedInfinity[I] <= Indeterminate
```

```
α <= α
```

The following example, regarded as a condition on `aVar`, can be passed to functions that use inequalities.

```
aVar < 23
```

Mathematica can also compare algebraic or irrational symbolic expressions using numerical techniques.

```
Sqrt[2] < Sqrt[3]
```

```
Pi > -2
```

```
I^I < E
```

The reason for the message generation in the last input was the internal use of numerical calculations. Inside a numerical calculation, we do not get an identically zero imaginary part for the $i^i = e^{-\pi/2}$ expression [95★], but instead get `0.0I`. `0.0I` is a complex number (head `Complex`), and it cannot be compared with a real number.

```
N[I^I, 50]
```

When two numbers cannot be compared because of the presence of small imaginary parts in internal numerical calculations, an error message is generated and the input is returned unchanged.

So the following example also generates a message.

```
I < 3 I
```

Often, *Mathematica* is presented with a chain of inequalities with several of the signs <, ≤, >, and ≥. Here is one inequality representation.

```
FullForm[a < b > c]
```

`Inequality[`$expression_1$`, ` $relation_1$`, ` $expression_2$`, ` $relation_2$`, …, ` $relation_n$`, ` $expression_{n+1}$`]`

or

$expression_1 > relation_1 > expression_2 > relation_2 > \cdots > relation_n > expression_{n+1}$

gives `True` if *Mathematica* can determine whether $expression_i\ relation_i\ expression_{i+1}$ holds for all $i = 1, \ldots n$. If the contrary can be established, it gives `False`.

Thus, we have three sets of comparisons for the following results.

```
{1 < 2 > 1, 2 <= 2 >= 2, 1 > 3 < 2}
```

Be aware that sometimes inequalities must be input as such directly.

```
Inequality[a1, Less, a2, Less, a3] // FullForm
```

```
InputForm[%]
```

Inputting the same inequality with "<" yields an expression with head `Less`.

```
a1 < a2 < a3
```

```
FullForm[%]
```

```
InputForm[%]
```

For some relations, expressions with head `Inequality` can evaluate to a logical combination of simpler inequalities.

```
Inequality[a1, Less, a2, Greater, a3]
```

```
FullForm[%]
```

If possible, an `Inequality` simplifies automatically.

```
1 < 2 < Z
```

The following example is also an `Inequality`.

```
1 <= 2 >= 5 // Hold // FullForm
```

Here are three further important commands that do not end with `Q` and which give truth values when its arguments are numbers.

`Positive[`*expression*`]`

gives `True` if *expression* is a positive number; otherwise, it gives `False`. If the truth value cannot be explicitly determined, `Positive[`*expression*`]` is returned unevaluated.

`Negative[`*expression*`]`

gives `True` if *expression* is a negative number; otherwise, it gives `False`. If the truth value cannot be explicitly determined, `Negative[`*expression*`]` is returned unevaluated.

`NonNegative[`*expression*`]`

gives `True` if *expression* is not a negative number; otherwise, it gives `False`. If the truth value cannot be explicitly determined, `NonNegative[`*expression*`]` is returned unevaluated.

Here is a test. In comparison to the predicates ending with `Q`, some of the following expressions remain unevaluated because *Mathematica* cannot determine their truth value uniquely.

```
testList = {2, -0.8, 0, 0.0, Pi, -E, -Sqrt[5], NaN, Infinity,
            0. I, 0``100, 3 + 0.I};

TableForm[Function[t, {Positive[t], Negative[t], NonNegative[t]},
                   Listable][testList],
          TableHeadings -> {testList,
          (* in bold *) StyleForm[#, FontWeight -> "Bold"]& /@
                   {"Positive", "Negative", "NonNegative"}},
          TableSpacing -> {1, 1}]
```

Note that 0 is neither positive nor negative. One purpose of `Positive` and `Negative` is not to determine whether a given number is positive or negative, but rather their use for the abstract definitions of properties with certain parameters. Here, for example, we want to make it known that `a` is positive.

```
Clear[a];
a /: Positive[a] = True;
```

```
??a
```

This information can now be used, for example, to define a case distinction in some routine.

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

■ 5.1.2 Boolean Functions for General Expressions

The following functions can be used to determine whether an expression matches a more general form.

`PolynomialQ[`*expression*`, {`x_1`, `x_2`, …, `x_n`}]`

gives `True` if *expression* is a polynomial in the variables x_1, x_2, …, x_n. If only one variable exists, the braces `{}` can be dropped.

Here is a simple example.

```
PolynomialQ[x^2 - 2 x + 3, x]
```

Here, it is important to note that this test can be applied to several variables.

```
polyYesNo = x^2 y^3 + 34 x^2 + 7 - Sin[z^3] x^34
```

In `z`, `polyYesNo` is not a polynomial.

```
PolynomialQ[polyYesNo, {x, y, z}]
```

However, in `x` and `y`, it is a polynomial.

```
PolynomialQ[polyYesNo, {x, y}]
```

In variables that are not present in an expression, the expression is considered to be a polynomial (the term *variable*^{0}).

```
PolynomialQ[polyYesNo, notPresentVariable]
```

In *Mathematica*, vectors are represented as lists.

```
vec = {111, 112, 113}
```

`VectorQ` determines whether an expression is a vector.

`VectorQ[`*expression*`]`

gives `True` if *expression* is a vector (whose elements are not lists).

We get the expected value `True` for `vec`.

```
VectorQ[vec]
```

For the following structure, we get `False` because the elements themselves have the head `List`.

```
VectorQ[{{1, 1}, {2, 2}, {3, 3}}]
```

However, despite the fact that the elements of the vector `{list[1, 1], list[2, 2], list[3, 3]}` are entries of `list` with more than one argument in the following expression, `list` is not `List`. `List` is a very special head in *Mathematica*.

```
VectorQ[{list[1, 1], list[2, 2], list[3, 3]}]
```

On the other hand, as soon as one `List` appears, we again get `False` as the truth value.

```
VectorQ[{list[1, 1], list[2, 2], List[3, 3]}]
```

Matrices in *Mathematica* are represented as vectors whose elements are vectors. The inner vectors correspond to the rows of the matrix (although *Mathematica* does not distinguish between row and column vectors; we return to this point in the next chapter).

```
mat = {{a11, a12, a13},
       {a21, a22, a23},
       {a31, a32, a33}}
```

MatrixQ[*expression*]

gives True if *expression* is a matrix, that is, a list of lists with the same length whose elements are not again lists.

The above mat is indeed a matrix.

```
MatrixQ[mat]
```

This example is also a matrix, although now the elements have depths 1

```
{{a[1, 1], a[1, 2], a[1, 3]},
 {a[2, 1], a[2, 2], a[2, 3]},
 {a[3, 1], a[3, 2], a[3, 3]}} // MatrixQ
```

If the elements have the head List, that is, the resulting object is a tensor of higher order, MatrixQ gives False.

```
{{{1, 1}, {1, 2}, {1, 3}},
 {{2, 1}, {2, 2}, {2, 3}},
 {{3, 1}, {3, 2}, {3, 3}}} // MatrixQ
```

However, when the elements of a matrix have multiple arguments and the head of the elements is not List, MatrixQ gives True.

```
{{list[1, 1], list[1, 2], list[1, 3]},
 {list[2, 1], list[2, 2], list[2, 3]},
 {list[3, 1], list[3, 2], list[3, 3]}} // MatrixQ
```

VectorQ and MatrixQ can also perform more general tests.

VectorQ[*expression*, *elementTest*]

gives True if *expression* is a vector such that the test *elementTest* is satisfied for each of its elements.

MatrixQ[*expression*, *elementTest*]

gives True if *expression* is a matrix such that the test *elementTest* is satisfied for each of its elements.

elementTest in the last two commands is a function that is applied to each element of *expression*. Only when the test returns true for all elements of the vector/matrix, True is returned. Thus, we could test for our vec and mat as follows.

```
{VectorQ[vec, NumberQ], VectorQ[vec, IntegerQ], VectorQ[vec, EvenQ]}
```

```
{MatrixQ[mat, NumberQ], MatrixQ[mat, IntegerQ], MatrixQ[mat, EvenQ]}
```

The test can be (and usually is) expressed in the form of a pure function.

```
{VectorQ[vec, (# > 0)&], VectorQ[vec, PrimeQ[#^2 - 1]&]}
```

It is also possible to test two or more expressions for “equality” or “nonequality”.

Equal[*expression*$_1$, *expression*$_2$, …, *expression*$_n$]

or

$expression_1$ == $expression_2$ == ⋯ == $expression_n$

gives `True` if *Mathematica* can determine that all of the expressions $expression_1$, …, $expression_n$ are identical. If it can be determined that this does not hold, it gives `False`. This explicit determination of the truth value is only possible when dealing with numeric expressions, strings, and identical symbols (lists are compared according to their list structure). If no truth value can be determined, *Mathematica* considers the above expression as a mathematical identity (in the sense of a condition on the variables of the corresponding routine, e.g., `Solve`, `DSolve`, and `FindRoot`), and returns the input.

To check whether two (or more) expressions are unequal, we use `Unequal`.

`Unequal[`$expression_1$`,` $expression_2$`, …,` $expression_n$`]`

or

$expression_1$!= $expression_2$!= ⋯ != $expression_n$

gives `True` if *Mathematica* can determine that no two of the expressions $expression_1$, …, $expression_n$ are identical. If it can be determined that this does not hold, it gives `False`. This explicit determination of the truth value is only possible when dealing with numeric expressions, strings, and identical symbols. If no truth value can be determined, the input is returned.

`Equal` and `Unequal` are not predicates ending with `Q`. But for numbers, strings, numeric expressions, and (nested) lists, they generically evaluate to `True` or `False`.

> `Equal` does not assign values as do `Set` and `SetDelayed`. Mathematical equalities are expressed with `Equal`.

We now test whether `a` has the value 2. Because we have not assigned any value to `a`, no decision can be made.

```
Remove[a];
a == 2
```

Performing this test does not change `a`, but if we had not used `a` so far in this *Mathematica* session, it would now be added to the list of symbols used.

```
??a
```

Once `a` has been assigned a numeric value, `Equal` and `Unequal` both deliver a result.

```
a = 2
```

```
{a == 2, a == 3, a != 2, a != 3}
```

The string `"b"` and the variable `b` are different objects.

```
Clear[b];
b == "b"
??b
```

```
(* assign value to b *)
b = "b";
{b == "b", b == "c", b != "b", b != "c"}
```

When comparing approximate numbers, only the significant digits are compared.

```
Clear[d, d1];
d = 5.85934859;
d1 = d + $MachineEpsilon;
d == d1
```

```
((1.0 + $MachineEpsilon/4)) == ((1.0 + $MachineEpsilon/3))
```

The difference between the last two numbers is identically zero.

```
((1.0 + $MachineEpsilon/4) - 1.0) -
((1.0 + $MachineEpsilon/3) - 1.0) // FullForm
```

Actually, the difference between two machine numbers (of size 1) can be around `$MachineEpsilon` so that `Equal` returns `True`. This behavior allows identification of numbers that arise from different calculations, but are "equal" (we discuss the much more stringent `SameQ` soon).

```
1.0 == 1.0 + 10 $MachineEpsilon
```

```
1.0 == 1.0 + 200 $MachineEpsilon
```

Depending on the absolute size of the number, smaller or larger deviations influence the result of comparisons with `Equal`. Adding something to a machine zero is often recognizable within machine arithmetic.

```
0.0 == 0.0 + 1/100 $MachineEpsilon
```

```
0.0 == 0.0 + 1/10^100 $MachineEpsilon
```

In the next input, the second part on the right-hand side becomes a high-precision number. Adding it to the machine number `0.0` results in the machine number `0.0`, which is identical to the left-hand side.

```
0.0 == 0.0 + $MinMachineNumber/10
```

By adding a small quantity to a much larger quantity, the size of their ratio determines if they are still considered equal.

```
100.0 == 100.0 + 1000 $MachineEpsilon
```

```
100.0 == 100.0 + 10000 $MachineEpsilon
```

Here is a similar example for a high-precision number. `e1` has 35 digits after the decimal point. Roughly speaking, for high-precision numbers, `Equal` does not take into account the last two digits.

```
Clear[e1, e2, e3];
e1 = 23.86784923634784599263894500083564995;
e2 = e1 + 10^-32;
e3 = e1 + 10^-33;
{e1 == e2, e1 == e3, e2 == e3}
```

The following four zeros are equal (in the sense of `Equal`) in spite of their different heads.

```
0 == 0.0 == 0.0 + 0.0 I == 0.0 I
```

In the following inequality (head `Inequality`), all elements must be different from each other.

```
1 != 2 != 3 != 4 != 1 != 5
```

Here, `1.`, is an approximative number and `1` is an integer, but they are not considered to be different when comparing them with `Unequal`.

```
1 != 2 != 3 != 4 != 1. != 5
```

```
1 != 1.
```

Now, we compare "explicitly identical" objects with one another. (Note the brackets and that, in the third element of the list of examples, `Equal` compares `Null` with `Null`.)

```
Clear[a, b, c, d, r];
a == a
```

```
b + c^d == b + c^d
```

```
(4;) == (5;)
```

```
(a = r; b = r) == (c = r; d = r)
```

```
Clear[a, b];
a + b == b + a
```

```
Hold[a + b] == Hold[a + b]
```

Here are some examples that do not evaluate to `True` or `False`. Inside `Hold`, no reordering takes place.

```
Hold[a + b] == Hold[b + a]
```

```
a + b == HoldPattern[a + b]
```

```
Indeterminate == Infinity
```

We repeat that only definitely comparable objects lead to `True` or `False`.

```
1 == 2
```

```
Integer == Symbol
```

```
"1" != 1
```

```
"a" == "aa"
```

In addition to comparing raw expressions like strings and numbers, there is one more case where `Equal` will not stay unevaluated, namely for nested `List`s. When the lengths or the depths of two lists do not agree, `False` is returned. Here is an example.

```
List[a, b] == List[a, b, c]
```

The last result happens although there exists a value for *c* such that the last comparison becomes an identity.

```
List[a, b] == (c = Sequence[]; List[a, b, c])
```

For heads other than `List`, no similar evaluation happens.

```
c =.
list[a, b] == list[a, b, c]
```

`True` and `False`, although symbols, are exceptions to the described rules about numbers, numeric quantities and strings.

```
True == False
```

`Equal` carries out numerical approximations. It is easy to verify within *Mathematica*'s high-precision arithmetic that the following two expressions are not the same.

```
(Sqrt[2] - 1)^2 == 3 + 2 Sqrt[2]
```

The following two expressions are (mathematically) the same. But using only numerical techniques, it is impossible to verify the equality. *Mathematica* issues a message and leaves the expression unevaluated. (We encountered the same message already in the last subsection.)

```
(Sqrt[2] - 1)^2 == 3 - 2 Sqrt[2]
```

For symbolic, nonnumeric expressions, `Equal` does not make any effort to prove (mathematical) equality.

```
a^2 + 2 a + 1 - (a + 1)^2 == 0
```

With only one argument in `Equal` or `Unequal`, the result (by definition) is `True`.

```
{Equal[onlyOneArg], Unequal[onlyOneArg], Equal[False], Equal[{}]}
```

An `Equal` structure can be given as the value of a variable via `Set` or `SetDelayed`.

```
immediateComparisonWithEqual = a1a == 2;
laterComparisonWithEqual := a1a == 2;
```

Here is the suppressed grouping.

```
FullForm[Hold[immediateComparisonWithEqual = a1a == 2]]
```

We now have the following values.

```
{immediateComparisonWithEqual, laterComparisonWithEqual}
```

`Equal` can produce a result after a value is assigned to `a1a`. (Note that in this case, `Set` and `SetDelayed` give the same result because, despite `Set`, the value of `immediateComparisonWithEqual` is not evaluated because no comparison can be made at the time `immediateComparisonWithEqual` is defined.)

```
a1a = 2
```

```
{immediateComparisonWithEqual, laterComparisonWithEqual}
```

To definitively decide if two quantities are identical, we use `SameQ`. As a ...`Q` function, it is a predicate.

`SameQ[`*expression*$_1$`,` *expression*$_2$`, …,` *expression*$_n$`]`

or

expression$_1$ `===` *expression*$_2$ `===` ⋯ `===` *expression*$_n$

gives `True` if *Mathematica* can determine that all of the expressions *expression*$_1$, …, *expression*$_n$ are identical. Otherwise, it gives `False`. `SameQ` also produces either `True` or `False`, even if the *expression*$_i$ are not numbers or strings, that is, they do not remain unevaluated.

While in general two expressions that are identical (in the sense of `SameQ`) are also equal (in the same of *Equal*), this is not the case for the symbol `Indeterminate`.

```
Indeterminate == Indeterminate
```

```
Indeterminate === Indeterminate
```

The reason for this potentially unexpected behavior is the fact the `Equal` expresses mathematical equality, and in a mathematical sense, one does not want 0^0 to be equal with $0/0$.

I similar remark hold for the quantity `ComplexInfinity`.

```
ComplexInfinity == ComplexInfinity
```

```
ComplexInfinity === ComplexInfinity
```

Thus, `SameQ[`*arg*`]` is essentially equivalent to `TrueQ[Equal[`*arg*`]]` (some small differences exist in the case that *arg* is an approximate number; we come back to this difference between `Equal` and `SameQ` in Chapter 1 of the Numerics volume [139✱]). A closely related test is `UnsameQ`.

`UnsameQ[`*expression*$_1$`,` *expression*$_2$`, …,` *expression*$_n$`]`

or

expression$_1$ =!= *expression*$_2$ =!= ⋯ =!= *expression*$_n$

gives `True` if *Mathematica* can determine that no two of the expressions *expression*$_1$, …, *expression*$_n$ are identical. Otherwise, it gives `False`. `UnsameQ` also produces either `True` or `False`, even if the *expression*$_i$ are not numbers or strings; that is, they do not remain unevaluated.

Now, we give some examples. Let us start by testing two machine numbers. In comparison to `Equal`, now the two machine numbers must agree with each other roughly within `$MachineEpsilon`.

```
1.0 === 1.0 + 2 $MachineEpsilon
```

```
1.0 === 1.0 + 0.5 $MachineEpsilon
```

And here two high-precision numbers are compared. Now, they must agree basically within all but the last of their digits.

```
N[1, 30] === N[1, 30] + 2 10^-30
```

```
N[1, 30] === N[1, 30] + 5 10^-30
```

Like in `Unequal`, all arguments have to be pairwise different to give `True` when the comparison is done with `UnsameQ`.

```
1 =!= 2 =!= 3 != 4 =!= 1. =!= 5
```

`SameQ` tests the structure of its arguments (taking into account the precision for numbers). It does not analyze their mathematical content; it is a purely structural operation.

```
Exp[-Pi/2] === I^I
```

```
a^2 + 2a b + b^2 === (a + b)^2
```

Similarly, a dummy integration variable makes a difference for `SameQ`. The following integral cannot be integrated in elementary functions.

```
Clear[x, ξ, y];
Integrate[x^x, {x, 0, y}]
```

Thus, the following unevaluated integrals are not considered the same because of the different dummy integration variable.

```
Integrate[x^x, {x, 0, y}] === Integrate[ξ^ξ, {ξ, 0, y}]
```

The following two pure functions act the same, but from a programming language standpoint they are different because they use different variables.

```
Function[x, x^2] === Function[ξ, ξ^2]
```

However, the following two expressions are identical after parsing; whether we input an expression in `FullForm` or in `InputForm`, or whatever, it has no effect on the internal representation.

```
Hold[1 + 1] === Hold[Plus[1, 1]]
```

The following example gives `False` because the internal form of `Hold[1 - 1]` is `Hold[Plus[1, - 1]]`.

```
Hold[Subtract[1, 1]] === Hold[1 - 1]
```

`1-I` and `Complex[1, 1]` are not the same expressions. `1-I` is `Plus[1, Times[-1, I]]`, which evaluates to the complex number (head `Complex`) `1-I`.

```
Hold[1 - I] === Hold[Complex[1, -1]]
```

```
Hold[1 - I] // FullForm
```

If we clear the value of `a` in the example considered above for `Equal`, we now get `False`.

```
Clear[a];
a === 2
```

But `Set` and `SetDelayed` work differently with `SameQ` than when compared with the corresponding examples, which use `Equal`.

```
Clear[a1a];
immediateComparisonWithSameQ = a1a === 2;
laterComparisonWithSameQ := a1a === 2;
```

Both now give `False` because `a1a` is not 2.

```
{immediateComparisonWithSameQ, laterComparisonWithSameQ}
```

But after assigning the value 2 to `a1a`, the value of `immediateComparisonWithSameQ` remains the same as before, whereas `laterComparisonWithSameQ` is recomputed.

```
a1a = 2;
{immediateComparisonWithSameQ, laterComparisonWithSameQ}
```

`Equal` is used for stating equality (in the sense of mathematical identities or conditions) in equations and for comparing numbers and strings. `SameQ` is used to test arbitrary expressions for equality.

In the next example, we first define a function $\Upsilon_i^k(z)$ iteratively. Then we use `FixedPoint` to calculate the limit $\lim_{k\to\infty} \Upsilon_i^k(z)$ for given starting values of i and z. We do this making use of a two-element list in `FixedPoint` with the first element being k and the second element being $\Upsilon_i^k(z)$ and we increase k at each step. We end the iteration when $\Upsilon_i^k(z)$ agrees with $\Upsilon_i^{k-1}(z)$ to all relevant digits.

```
ϒ[i_Integer, 0, z_] := (1 + z/2^i)^(2^i)

ϒ[i_Integer, k_, z_] := ϒ[i, k, z] =
      (2^k ϒ[i + 1, k - 1, z] - ϒ[i, k - 1, z])/(2^k - 1)

exp[i_][z_] := FixedPoint[{#[[1]] + 1, ϒ[i, #[[1]] + 1, z]}&,
                          {0, ϒ[i, 0, z]},
                          SameTest :> (#1[[2]] === #2[[2]]&)]
```

For all i and z, the limit of the above iteration is the exponential function exp(z) [146✱].

```
{exp[-10][N[1 + I, 21]], exp[+10][N[1 + I, 21]], Exp[N[1 + I, 20]]}
```

We now present several other important structural Boolean functions.

`OrderedQ[`*expression*`[`*subExpression*$_1$`,` *subExpression*$_2$`, …,` *subExpression*$_n$`]]`
gives `True` if the expressions *subExpression*$_1$, …, *subExpression*$_n$ are in canonical order.

Here are two simple examples with lists.

```
{OrderedQ[{1, 2, 3, aaa, bbb, ccc}],
 OrderedQ[{1, 2, 3, aaa, bbb, ccc, 4}]}
```

Functions with the attribute `Orderless` will reorder their arguments. If two arguments are ordered is tested with `OrderedQ`.

```
SetAttributes[orderlessFunction, Orderless];
orderlessFunction[1, 2, 3, aaa, bbb, ccc, 4]
```

To test whether a given object is contained in another, we use `MemberQ`.

> `MemberQ[`*expression*`,` *subExpression*`,` *level*`]`
>
> gives `True` if *object* appears in *subExpression* at the level *level*, and it otherwise gives `False`. If *level* does not appear, it is taken to be 1. The usual level specifications hold (see Chapter 2).

Here is a somewhat more detailed example (to review `Level`). We define the expression `object2`.

```
object2 = Sin[Log[3 σ x/2] 56 Cos[r^2]]
```

The entire argument of `Sin` appears in level 1.

```
MemberQ[object2, Log[3 σ x/2] 56 Cos[r^2]]
```

But `r` appears in the root in level `{-1}`.

```
{MemberQ[object2, r], MemberQ[object2, r, -1]}
```

Note that 3 does not appear at all in `object2`.

```
MemberQ[object2, 3, Infinity]
```

Together 3 and 2 in 3/2 form one rational number (see Subsection 2.3.3).

```
MemberQ[object2, 3/2, Infinity]
```

Analogous to some commands from Chapter 2 (like `Level` and `Position`), `MemberQ` has the option `Heads`.

```
Options[MemberQ]
```

```
MemberQ[Sin[Sin[3]], Sin, {0, Infinity}]
```

To see the `Sin` in `Sin[Sin[3]]`, we must use the option setting `Heads -> True`.

```
MemberQ[Sin[Sin[3]], Sin, {0, Infinity}, Heads -> True]
```

The opposite of `MemberQ` is accomplished with `FreeQ`.

> `FreeQ[`*expression*`,` *subExpression*`,` *level*`]`
>
> gives `True` if *subExpression* does not appear in *expression* at the level *level*, and it gives `False` otherwise. If *level* does not appear, it is taken to be `Infinity`. The usual level specifications hold.

The integer 3 is not contained in `object2`.

```
FreeQ[object2, 3]
```

`r` appears in `object2` but not at level 1.

```
FreeQ[object2, r]
```

An expression itself is also recognized by `FreeQ`.

```
FreeQ[r, r]
```

```
FreeQ[r, r, {1}]
```

Note the difference in the default values for the levels in the third arguments of `MemberQ` and `FreeQ`.

```
MemberQ :1
FreeQ   :Infinity
```

The second argument of `FreeQ` is considered purely from the standpoint of structure and not (mathematical) content. We give examples with definite integration and pure functions to illustrate this fact.

```
Clear[p, Σ, 𝟙];
```

```
{FreeQ[Function[p, p^2], p],
 FreeQ[Integrate[Σ[𝟙], {𝟙, 0, p}], 𝟙]}
```

Finally, we present the last two tests to be treated here, `ValueQ` and `AtomQ`.

`ValueQ[`*expression*`]`

gives `True` if *expression* has a value, and it gives `False` otherwise.

`AtomQ[`*expression*`]`

gives `True` if *expression* is an atomic object (i.e., if it does not contain any subexpressions, like a number, symbol, or string). Otherwise, it gives `False`.

The following exotic variable surely has not yet been assigned a value.

```
ValueQ[abcdefghijklmnopqrstuvwxyz]
```

Now, we assign it an equally exotic value.

```
abcdefghijklmnopqrstuvwxyz = zyxwvutsrqponmlkjihgfedcba
```

Now, `ValueQ` gives `True`.

```
ValueQ[abcdefghijklmnopqrstuvwxyz]
```

Here is the ownvalue of `abcdefghijklmnopqrstuvwxyz`.

```
OwnValues[abcdefghijklmnopqrstuvwxyz]
```

The following `atomQ` performs like `AtomQ`.

```
Remove[atomQ];
atomQ[x_] := Level[x, {0, Infinity}, Heads -> True] === {x};
```

```
{AtomQ[1], AtomQ[-t], AtomQ[2 + 3 I], AtomQ[1/r],
 AtomQ[Hold[1 + 1]], AtomQ[C[]]}
```

```
{atomQ[1], atomQ[-t], atomQ[2 + 3 I], atomQ[1/r],
 atomQ[Hold[1 + 1]], atomQ[C[]]}
```

Note that `atomQ` is a rough approximation to the built-in `AtomQ`, but it might not work properly if its argument has the head `Unevaluated` because, in this case, `Level` evaluates its argument.

```
{AtomQ[Unevaluated[1 + 1]], atomQ[Unevaluated[1 + 1]]}
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

■ 5.1.3 Logical Operations

The commands for the classical logical operations are given as follows.

`Not[`*expression*`]`
or
`!`*expression*

gives `True` if *expression* is a false statement, and it gives `False` if *expression* is a true statement. If the truth value cannot be determined explicitly, the statement is interpreted as a statement that should hold.

`Or[`$expression_1$`,` $expression_2$`, …,` $expression_n$`]`
or
$expression_1$ `||` $expression_2$ `||` ⋯ `||` $expression_n$

gives `True` if *Mathematica* can determine that at least one of the $expression_i$ is true. It gives `False` if they are all false. If neither truth value can be computed, *Mathematica* interprets $expression_1$ `||` $expression_2$ `|| … ||` $expression_n$ as a statement that should hold.

`And[`$expression_1$`,` $expression_2$`, …,` $expression_n$`]`
or
$expression_1$ `&&` $expression_2$ `&&` ⋯ `&&` $expression_n$

gives `True` if *Mathematica* can determine that all $expression_i$ are true. It gives `False` if at least one of them is explicitly false. If no truth value can be determined, *Mathematica* interprets $expression_1$ `&&` $expression_2$ `&& … &&` $expression_n$ as a statement that should hold.

`Xor[`$expression_1$`,` $expression_2$`, …,` $expression_n$`]`

gives `True` if *Mathematica* can determine whether an odd number of the $expression_j$ are true, and it gives `False` if it can determine that an even number are true. If neither of these two truth values can be determined, *Mathematica* treats `Xor[`$expression_1$`,` $expression_2$`, …,` $expression_n$`]` as a statement that should hold.

(Be aware that in `!`*expression* in the beginning of an *Mathematica* input, the `!` is interpreted as a shell escape and *expression* will be sent to the operating system; this can be avoided by using `(!`*expression*`)`.)

Here are some examples. Is 4 < 5 and 567876 an integer or is 3 < 0 and 456 a prime?

```
((4 < 5) && IntegerQ[567876]) || (3 < 0 && PrimeQ[456])
```

Is 2 < 5 and 3 / 5 not an integer and −2 < 0?

```
(2 < 5) && (!IntegerQ[3/5]) && (-2 > 0)
```

In Chapter 4, we mentioned that the computation of the arguments of logical functions proceeds in a nonstandard way. Calculations are carried only far enough to make a decision. Thus, for example, the meaningless statements in the second and third arguments of the following `Or` expression remain untouched, and no error message is generated.

```
1 < 2 || I < 2 I || Sin[1, 2, 3, 4, 5, 6, 7, 8]
```

Because `And` and `Or` have the attributes `HoldAll`, some of the arguments of `And` and `Or` might never be evaluated.

```
Attributes[And]

Attributes[Or]
```

For multiple nested logical expressions, the `LogicalExpand` command is important.

`LogicalExpand[`*expression*`]`

applies the logical distributive laws to simplify nested expressions in *expression* so that the result contains only expressions at a single level.

Here, we simplify an expression consisting of three parts combined with "or", each of which contains several subexpressions.

```
LogicalExpand[(!IntegerQ[ν] && EvenQ[τ]) ||
                    (ϵ < ω && ρ >= Δ) || (!ξ && β < σ)]
```

To help interpret the result, consider the following example.

```
{!IntegerQ[ν], EvenQ[τ]}
```

In the larger result above, `&&` and `||` appear next to each other. Here is the grouping used in such expressions.

```
FullForm[𝔸 || 𝔹 && ℂ]
```

`And` has higher grouping precedence than `Or`.

Thus, a difference exists between `False && False || True` and `False && (False || True)`.

```
t = True; f = False;
{f && f  || t, f && (f || t), (t && t) || t, f && (t || t)}
```

The same grouping applies in the more general infix form.

```
a ~ And ~ b ~ Or ~ c // FullForm
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

■ 5.1.4 Control Structures

In addition to the logical functions introduced in the last subsection, some other functions depend on truth values. The best known of these are the control structures used in all programming languages. Let us start with `If`.

`If[`*test*`, `*then*`, `*else*`, `*neither*`]`

gives the result *then* if *Mathematica* can determine that the test *test* is true. It gives the result *else* if the test *test* is false. If the test *test* cannot be established to be either true or false, it gives the result *neither*. The last or the last two arguments can be dropped.

If the last argument in `If[`*test*`, `*then*`, `*else*`, `*neither*`]` is not present, and *Mathematica* is not able to find the truth value of *test*, the entire `If` expression is returned unchanged.

Thus, the 5 is not substituted in the last argument of the following expression.

```
she = 5;
If[she > he, who, she]
```

But the `she` in the first argument of `If` gets evaluated. The reason is the `HoldRest` attribute of `If`.

```
Attributes[If]
```

This means that the first argument of `If` gets evaluated in any case. All other arguments will not be evaluated in the beginning. Only when the first argument is explicitly `True` or `False` will the second or third argument be evaluated. `If` is a programming construct. `If` should *not* be used to model a step function.

```
HeavisideTheta[x_] = (* bad idea *) If[x > 0, 1, 0];
```

To check, we plot it.

```
Plot[HeavisideTheta[x], {x, -2, 2},
     Axes -> True, AxesOrigin -> {-2.2, -0.2},
     PlotStyle -> {Thickness[0.02]}]
```

The built-in function `UnitStep` (discussed in Chapter 1 of the Symbolics volume [140✱]) is much more suited for the construction of piecewise functions.

Like `If`, the related command `Which` also depends on the calculation of truth values. It is the obvious generalization of `If`.

`Which[`*test*$_1$`,` *then*$_1$`,` *test*$_2$`,` *then*$_2$`,` …`,` *test*$_n$`,` *then*$_n$`]`

gives the result *then*$_i$, where the test *test*$_i$ is the first one that can be determined to be true. If one of the tests *test*$_i$ is indefinite, this expression remains unevaluated. If all of the tests *test*$_i$ are determined to be false, `Null` is returned.

When we want to look for some elements from an expression (from a set) for which a special criterion is true, we can use `Select`.

`Select[`*expression*`,` *criterion*`,` *howMany*`]`

gives the first *howMany* parts of the first level of *expression* for which the criterion *criterion* is true. If the integer *howMany* is not present, all subexpressions are found. The head of the resulting expression is the same as that of *expression*. If the last argument is absent, all subexpressions that fulfill *criterion* will be returned.

Here are a few simple examples of `Which` and `Select`.

```
Which[1 > 3, 1, 2 > 3, 2, 3 > 3, 3, 4 > 3, 4, 5 > 3, 5]
```

```
Which[False, 1, False, 2, True, 3, False, 4]
```

Now, no case matches and the result is `Null`.

```
Which[5 == 6, m] // FullForm
```

The truth value of the first argument cannot be determined. As a result, the whole `Which` returns unevaluated. (The same happens if the truth value of any evaluated odd numbered argument cannot be determined.)

```
Which[undecided, 1, Print["I got evaluated!"]; False, 2, True, 3]
```

```
Select[{3, i, 8 p + Sin[3], 689 h, 𝕘, 33 I, 4r}, NumberQ]
```

Here, the head of the first argument of `Select` is `Plus`.

```
Select[3 + i + 8 p + Sin[3] + 689 h - 𝕘 + 33 I + 4r, NumberQ]
```

The following example leads to an error message because `Sin` should have just one argument. However, it illustrates the effect of `Select`. After the expression, `Sin[0.0, 3 E, Pi, False, 𝔾]` has generated an error message and remains unevaluated because no built-in rules exist for `Sin` with multiple arguments. Then, `Select` goes into effect giving `Sin[0.0]`, which evaluates to the result `0.0`.

```
Select[Sin[0.0, 3 E, Pi, False, 𝔾], NumberQ]
```

As in other programming languages, a calculation can be repeated based on a test using `While` and `For`.

`While[`*test*`,` *toDo*`]`

repeats the computation of the test *test* and evaluates the expression *toDo* as long as the test *test* gives `True`.

`For[`*start*`,` *test*`,` *step*`,` *toDo*`]`

begins with evaluating the expression *start*, and then repeats the computation of the test *test*, followed by the evaluation of evaluates the expressions *toDo* and *step* as long as the test *test* gives `True`.

Here is a list of different things.

```
testList = {1, 2, 3, 4, 5, 6, α, β, γ, δ, 1, 2, 3, 4, Pi, I};
```

Using `While`, we print out the entries that are smaller than 5 until we find one which is not smaller than 5.

```
i = 0;
While[i = i + 1; testList[[i]] < 5, Print[testList[[i]]]]
```

Here is a similar example using `For`.

```
For[i = 1, testList[[i]] < 5, i = i + 1, Print[testList[[i]]]]
```

Now, we give a more interesting example involving `While`: how many successive terms of the sequence $a_k = 4\,180\,566\,390\,k + 8\,297\,644\,387$ are prime numbers (see [59✱], [44✱], [104✱], [150✱], [151✱], [114✱], and [147✱])?

```
k = -1;
While[k = k + 1; PrimeQ[4180566390 k + 8297644387],
      CellPrint[Cell[TextData[{"∘ For ",
                       Cell[BoxData[FormBox["k = " <> ToString[k],
                                            TraditionalForm]]],
                       ", the resulting number ",
                       ToString[4180566390 k + 8297644387],
                       " is prime."}], "PrintText"]]]
```

We make one remark concerning the use of `While` and `For`. Constructions that use `For`, `While`, and `Do` in other programming languages can often be implemented in *Mathematica* in a cleaner, more elegant, and faster-executing way using list operations like `Map`, `Thread`, (all to be discussed in the next chapter) and so on and `Fold`, `FoldList`, `Nest`, `NestList`, `FixedPointList`, and `FixedPoint` from Chapter 3. We will encounter many such examples in the following chapters.

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

■ 5.1.5 Piecewise Functions

In Chapter 2, we discussed the elementary functions like trigonometric functions and their inverses. In Chapter 3 of the Symbolics volume, we will discuss the special functions, like Gamma and Bessel functions. A class of functions that are useful for many practical (modeling) problems are piecewise defined functions. In the last subsection, we discussed the programming construct `If` and used it in a very simple (and not recommended) example to build up a piecewise-defined function.

`Piecewise[{{`*value*$_1$`,` *condition*$_1$`}, {`*value*$_2$`,` *condition*$_2$`}, ...},` *defaultValue*`]`

is a piecewise defined function with value *value*$_1$ when the condition *condition*$_1$ holds, with value *value*$_2$ ... and with value *defaultValue* in case none of the conditions is fulfilled.

Observe that, in distinction to functions like `If`, that the order is the value and then the condition to follow more closely the traditional notation used for piecewise defined functions.

Here is a simple piecewise defined function.

```
pw1[x_] = Piecewise[{{1 - (x + 2)^2, x < -2}, {1, x < 0}, {2 - x, 0 < x < 2
```

Piecewise functions have a characteristic formatting in `TraditionalForm`.

```
pw1[x] // TraditionalForm
```

Here this function is shown.

```
Plot[pw1[x], {x, -3, 3}, PlotRange -> All, Frame -> True, Axes -> False,
     PlotStyle -> {Hue[0]}]
```

In many respects, piecewise-defined functions behave like any other built-in mathematical function. For instance, a piecewise-defined function can be differentiated.

```
D[pw1[x], x]
```

Be aware that the value of the derivative at points where the functions is discontinuous is `Indeterminate`. But at the point $x = -2$, the function is continuous, its left- and right-sided derivatives exists and are identical. As a result, the derivative has a value there. (This last remark holds only for univariate functions; in the multivariate case, no detailed analysis of the degree of continuity at region boundaries is carried out.)

Piecewise functions can also be integrated.

```
Integrate[pw1[x], x]
```

```
Integrate[pw1[x], {x, -3, y}, Assumptions -> Element[y, Reals]]
```

The following integral fails because it extends over an infinite domain.

```
Integrate[FractionalPart[x] Exp[-x], {x, 0, Infinity}]
```

A powerful function to canonicalize piecewise-defined functions is `PiecewiseExpand`.

`PiecewiseExpand[`*expression*`]`
combines *expression* containing arithmetic operations of piecewise functions into one piecewise function.

Here is a rational function of the above piecewise function `pw1[`*x*`]` and a version of it with a translated argument. Contrary to *Mathematica*'s overall assumption that all occurring variables in an expression are generic complex values, the implicit assumption that α is real is made. This happens because the conditions of the piecewise functions contain comparison functions.

```
PiecewiseExpand[(2 pw1[x] + pw1[x]^2)(1 + pw1[x - α]^3)]
```

Here the resulting piecewise function is shown over the x,α-plane.

```
Plot3D[Evaluate[%], {x, -3, 3}, {α, -3, 3}, PlotPoints -> 60]
```

Also, compositions of piecewise -defined functions are written as one piecewise function by applying the function `PiecewiseExpand`.

```
PiecewiseExpand[pw1[pw1[x] + Sin[pw1[x]]]] //
                  (* avoid long lines *) InputForm
```

Here is another piecewise-defined function. This time, we have complex arguments in mind.

```
pw2[z_] = Piecewise[{{-1, Im[z] < 0}}, 1]
```

No inference that the arguments are real is made this time.

```
pw2[z] pw2[z - w] pw2[z] pw2[z + w] // PiecewiseExpand
```

We can give additional assumptions using the `Assumptions` option. We will discuss this in more detail in the beginning of Chapter 1 of the Symbolics volume.

```
PiecewiseExpand[%, Assumptions -> Element[w, Reals]]
```

In addition to `Piecewise` itself, many other functions are rewritten as piecewise functions (head `Piecewise`) by `PiecewiseExpand`. The following table shows the function on the left-hand side and its piecewise equivalent on the right hand side. We assume $-2 < x < 2$ and $1 < y < 2$.

```
{#, (* use assumptions and rewrite through Piecewise *)
    Assuming[-2 < x < 2 && 1 < y < 2, PiecewiseExpand[#]]}& /@
(* list of functions to be rewritten through Piecewise *)
{Abs[x], Boole[x], Ceiling[x], Floor[x],
 FractionalPart[x], If[x > 1, 2, 1], IntegerPart[x],
 Max[x, y], Min[x, y], Mod[x, y], Quotient[x, y], Round[x], Sign[x],
 Switch[x < 0, -1, x > 1, 2, True], UnitStep[x, y],
 Which[x < 0, -1, x > 1, 2, True, 0]} // TableForm
```

Above, we discussed Boolean functions. *Mathematica* has the built-in function `Boole` too.

`Boole[`*expression*`]`

represents the value 1 if expression evaluates to `True` and 0 else.

In the next input, `Boole` evaluates to 1.

```
Boole[True]
```

For symbolic x and y, we can use the expression `Boole[x^2 + y^2 < 1]` to represent a unit disk. The expression does not evaluate nontrivially.

```
Boole[x^2 + y^2 < 1]
```

But we can use such-type expression in other functions to specify geometric domains. The next input calculates the area of the unit disk.

```
Integrate[Boole[x^2 + y^2 < 1],
          {x, -Infinity, Infinity}, {y, -Infinity, Infinity}]
```

And here is the area of a unit sphere.

```
Integrate[Boole[x^2 + y^2 + z^2 < 1],
          {x, -Infinity, Infinity}, {y, -Infinity, Infinity},
          {z, -Infinity, Infinity}]
```

The function `PiecewiseExpand` convert the function `Boole` into a `Piecewise` function.

```
PiecewiseExpand[Boole[x^2 + y^2 < 1]]
```

We end with a small application of piecewise functions.

Mathematical Remark: Endpoint Distance Distribution of Random Flights

Consider a random stepwise flight of a particle in $\mathbb{R}^3$. The particle starts at the origin and each flight step has unit length and is taken in a randomly chosen direction. The probability $p_n(r)$ that the particle is found at a distance r from the origin after n steps is given by the following integral [26✱]

$$p_n(r) = \frac{1}{2\pi^2 r}\int_0^\infty \sin(\rho\, r)\left(\frac{\sin(\rho)}{\rho}\right)^n \rho\, d\rho.$$

This is the probability definition through a definite integral.

```
p[n_][r_] := Integrate[Sin[ρ r] (Sin[ρ]/ρ)^n ρ, {ρ, 0, Infinity},
                       Assumptions -> r > 0]/(2 Pi^2 r)
```

Mathematica cannot carry out the integral for symbolic n, but returns values for the integrals for concrete $n > 1$ that return the absolute value and the signum function.

```
p[n][r]
```

```
Table[p[n][r], {n, 2, 6}]
```

A more easily readable result is obtained by writing the integration results as a piecewise function.

```
Table[PiecewiseExpand[p[n][r], r > 0], {n, 2, 6}]
```

We see that, with the exception of $p_2(r)$, all higher $p_n(r)$ are well behaved at the origin. Obviously, after one step, the distance of the particle from the origin is 1 and we have $p_1(r) = \delta(r-1)$. This Dirac delta function cannot be the result of a classically convergent integral. So evaluating `p[1][r]` generates a message and stays unevaluated. (`p[1][r]` is effectively a Fourier sin transform that results in a generalized function.)

```
p[1][r]
```

Piecewise defined functions can largely be used as any named function (like `Sin`). They can be integrated and differentiated; they can appear in equations and inequations, and so on. Next, we check the normalization of the resulting probabil ity distributions. We have $\int_{\mathbb{R}^3} p(|\mathbf{r}|)\, d^3\mathbf{r} = 1$.

```
Table[4 Pi Integrate[p[n][r] r^2, {r, 0, Infinity}], {n, 2, 8}]
```

And here is the average distance of the particle after the n flight steps.

```
Table[4 Pi Integrate[r p[n][r] r^2, {r, 0, Infinity}], {n, 2, 8}]
```

```
N[%]
```

We end with plots showing the resulting distributions. $p_2(r)$ has a jump at $r = 2$ and $p_3(r)$ has a kink at $r = 1$. All other distributions seem to be smooth functions.

```
Plot[Evaluate[Table[p[n][r], {n, 2, 10}]], {r, 0, 10},
     PlotRange -> {0, 0.05}, Frame -> True, Axes -> False,
     PlotStyle -> Table[Hue[k/10], {k, 0, 9}]]
```

We see the piecewise character of the resulting distributions by plotting the first few derivatives with respect to r. We see that each derivative reveals a jump discontinuity at one more of the $p_n(r)$.

```
With[{ps = Table[PiecewiseExpand[p[n][r], r > 0], {n, 2, 10}]},
Show[GraphicsArray[
  (* make table of plots of derivatives *)
  Table[Plot[Evaluate[D[ps, {r, k}]], {r, 0, 10},
         PlotRange -> {-0.05, 0.05}, Frame -> True,
         FrameTicks -> False, DisplayFunction -> Identity,
         Axes -> False, PlotStyle -> Table[Hue[k/10], {k, 0, 9}]],
        {k, 5}]]]]
```

Using the function `Reduce` (to be discussed in the Symbolics volume), we can also easily calculate the position of the inflection points of the distribution curves. Most of these points are solutions of irreducible polynomials, which are returned as `Root`-objects (see Chapter 1 of the Symbolics volume [140✱]).

```
Table[{n, Reduce[D[PiecewiseExpand[p[n][r], r > 0], {r, 2}] == 0 &&
                 0 < r < n, r]}, {n, 5, 10}]

N[%]
```

For large n, we have $p_n(r) \approx (2\,\pi\,n/3)^{-3/2}\exp\left(-3\,r^2\big/(2\,n)\right)$. The following graphics shows this asymptotic expression and the exact curve for $n = 100$. We start with calculating $p_{100}(r)$. It is a quite large expression.

```
p100[r_] = PiecewiseExpand[p[100][r], r > 0];

{ByteCount[p100[r]], LeafCount[p100[r]]}
```

Here is a glimpse on the first r-interval $0 \leq r < 2$.

```
(* shortened form of the exact expression *)
Short[p100[r][[1, 1]], 12]

(* low precision numericalization *)
N[Expand[p100[r][[1, 1]]], 2]
```

Here is a plot of $p_{100}(r)$ and the asymptotic expression for $n = 100$. We use high-precision arithmetic to calculate the values of $p_{100}(r)$ because the large powers of r in the resulting expression would cause an excessive loss of precision for machine numbers and no correct curve could be plotted. The right plot shows the difference between the exact curve and the large n approximation.

```
p100HP[r_?InexactNumberQ] := N[p100[Rationalize[r, 0]], 20]

With[{n = 100},
 Show[GraphicsArray[
   Plot[(* asymptotic value and exact expression *)
        Evaluate[# @@ {1/(2Pi n/3)^(3/2) Exp[-3 r^2/(2n)],
                     p100HP[r]}], {r, 0, 30},
        PlotRange -> All, Frame -> True,
        Axes -> False, DisplayFunction -> Identity,
        PlotStyle -> {{Thickness[0.02], GrayLevel[0.8]}, Hue[0]}]& /@
        (* show both curves and their difference *) {List, Subtract}]]]

Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

5.2 Patterns

■ 5.2.1 Patterns for Arbitrary Variable Sequences

Before discussing more complicated pattern recognition in *Mathematica*, for self-containedness, we recall the patterns already discussed in Subsection 3.1.1.

`Blank[]` or `_`
is a pattern for an arbitrary *Mathematica* expression.

`Blank[`*head*`]` or `_`*head*
is a pattern for some arbitrary *Mathematica* expression with head *head*.

`Pattern[`*x*`, Blank[]]` or *x*`_`
is a pattern for some arbitrary *Mathematica* expression named *x*.

`Pattern[`*x*`, Blank[`*head*`]]` or *x*`_`*head*
is a pattern for some arbitrary *Mathematica* expression named *x* with head *head*.

We have already used patterns in functions. The patterns above allow the definition of functions for a fixed number of arguments. Here is a simple example.

```
f[x_] := x^x;
f[3]
```

But nothing happens in the next two inputs. The pattern does not match.

```
f[]
```

```
f[1, 2, 3]
```

To match the last input, we would need a pattern like `f[x_,y_,z_]`.

Often, it is not known how many arguments a function (e.g., `Plus`) will be given, or we may want to define a function only for certain classes of arguments, which may differ in other ways than by just their heads. We now discuss these possibilities.

We already defined the following factorial function.

```
fac[1] = 1; fac[n_] := fac[n - 1] n
```

For noninteger arguments, this definition leads to an infinite loop.

```
fac[38/11] // Shallow[#, 4]&
```

This problem can be avoided by using a construction that takes into account different patterns.

```
Clear[fac];
fac[1] = 1;
fac[n_Integer] := fac[n - 1] n
```

Because `fac` was defined only for integers, `fac[38/11]` now remains unevaluated.

```
fac[38/11]
```

For negative integers, this definition still fails. (We will discuss how to test for this case in a moment.) By testing the head only, it is impossible to distinguish between positive and negative exact integers.

```
fac[-2]
```

`Blank` stands for one occurrence of any expression, but some expression must exist in that position for the pattern to match. The function `fSomething` evaluates nontrivially if called with two arguments.

```
fSomething[_, _] := 555
```

With only one argument, no definition is matched.

```
fSomething[t]
```

For two arbitrary arguments, we always get 555.

```
fSomething[t, τ]

fSomething[-38/11, 0]
```

Now, we try `fSomething` with three arguments; we gave no definition for this pattern.

```
fSomething[t, τ, 𝕥]
```

For functions with more than one argument, we can use `BlankSequence[]`.

`BlankSequence[]`
  or
`__`
  is a pattern standing for a sequence of arbitrary *Mathematica* expressions with at least length 1.

Here is a definition of an analog of `fSomething`, which works for one or more than one argument.

```
Clear[𝔽];

𝔽[__] := 555;

{𝔽[], 𝔽[t], 𝔽[t, t], 𝔽[t, t, t], 𝔽[t, t, t, t]}
```

The analogous construction that takes into account heads is `BlankSequence[`*head*`]`.

```
FullForm[argument__headOfArgument]
```

`BlankSequence[`*head*`]`
  or
`__`*head*
  is a pattern standing for a sequence of arbitrary *Mathematica* expressions with at least one element, each of which has the head *head*.

All patterns discussed until now required at least one argument. The case of no argument or some arguments is covered by `BlankNullSequence`.

`BlankNullSequence[]`
  or

`___`

is a pattern standing for a sequence of arbitrary *Mathematica* expressions, including those of length 0, that is, for no expression.

`BlankNullSequence[`*head*`]`

or

`___`*head*

is a pattern standing for a sequence of arbitrary *Mathematica* expressions, including those of length 0 (i.e., no expression), each of which has the head *head*. If no expression is present, it automatically has the head *head*.

Here is a definition for our function from above that also gives 555 without an argument.

```
𝔽𝔽[___] := 555
```

```
{𝔽𝔽[], 𝔽𝔽[t], 𝔽𝔽[t, t], 𝔽𝔽[t, t, t], 𝔽𝔽[t, t, t, t]}
```

Now, we define yet another function, this time requiring the arguments to have the head `Symbol`. (Note that all arguments must have this head, and “no argument” is assumed to automatically have this head.)

```
𝔽𝔽𝔽[___Symbol] := 555;
```

```
{𝔽𝔽𝔽[], 𝔽𝔽𝔽[t], 𝔽𝔽𝔽[t, t], 𝔽𝔽𝔽[t, t, t], 𝔽𝔽𝔽[t, t, t, t]}
```

```
{𝔽𝔽𝔽[], 𝔽𝔽𝔽[1], 𝔽𝔽𝔽[1t], FF𝔽𝔽𝔽F[1t, 1t],
 𝔽𝔽𝔽[1, t, t], 𝔽𝔽𝔽[1, 2, t, t]}
```

We can determine if a pattern matches a certain expression using the function `MatchQ`.

`MatchQ[`*expression*`, `*pattern*`]`

returns `True` if the pattern *pattern* matches the expression *expression* and `False` otherwise.

Here are four patterns that match a four-argument function `f`.

```
MatchQ[f[1, 2, 3, {4, 5}], f[_, _, _, _]]
```

```
MatchQ[f[1, 2, 3, {4, 5}], f[_, _, _, _List]]
```

```
MatchQ[f[1, 2, 3, {4, 5}], f[_, __]]
```

```
MatchQ[f[1, 2, 3, {4, 5}], f[___]]
```

But not each of the four arguments has the head `List` in the following input.

```
MatchQ[f[1, 2, 3, {4, 5}], f[___List]]
```

The associated named patterns can be constructed using `Pattern`.

`Pattern[`*name*`, `*pattern*`]`

or

name`:`*pattern*

represents the pattern *pattern*, and the pattern is named *name*. If no confusion is possible, the colon can be left out.

In the following example, the function Ƒ𝔽 gets called with three arguments. All arguments together match the pattern ξ.

```
Ƒ𝔽[ξ:x___] := 𝔽Ƒ[ξ]
```

```
ƑF[1, 2, 3]
```

The use of the colon allows the hierarchical grouping of patterns and the naming of more complex patterns. Here, the colon allows grouping of the entire expression `{b_, c_}`, which already contains two patterns, to form a new one called `a`.

```
ϒ[a:{b_, c_}] := {a, b, c}
```

The pattern realization for the following input is given by `b` → `{2}`, `c` → `{1}` and `a` → `{1, 2}`.

```
ϒ[{1, 2}]
```

The whole left-hand side of an assignment (or a rule) can be a pattern, as in the next example, in which `pat` (=*pat*) is the whole expression.

```
Clear[p, pat, x, y, u];

u/: pat:p_[u, x_] := {p, x, Hold[pat]}

p[u, y]
```

We have the following typical possibilities for patterns in a sequence of arguments.

- `Pattern[x, Blank[]]` or *x*_ stands for an object named *x*.
- `Pattern[x, Blank[`*head*`]]` or *x*_*head* stands for an object with head *head* named *x*.
- `Pattern[x, BlankSequence[]]` or *x*__ stands for at least an object named *x*.
- `Pattern[x, BlankSequence[`*head*`]]` or *x*__*head* stands for at least one object named *x*, all with head *head.*
- `Pattern[x, BlankNullSequence[]]` or *x*___ stands for zero or more objects named *x*.
- `Pattern[x, BlankNullSequence[`*head*`]]` or *x*___*head* stands for zero or more objects named *x*, all with head *head.*

All of these patterns can be used for function definitions with `Set` and `SetDelayed`, for replacement patterns with `Rule` and `RuleDelayed`, and also for commands such as `Cases`, `DeleteCases`, and `MatchQ` (see below).

Note that `BlankNullSequence` is not a more "general" pattern (in the order of the rules associated with a `Symbol`) than is `BlankSequence` or `Blank`. We can demonstrate this in the following example. In the following two inputs, the rules are not reordered.

```
Clear[s];
s[_] = 1;
s[__] = 2;
s[___] = 3;
s[1]

??s

Clear[s];
s[___] = 3;
s[__] = 2;
s[_] = 1;
s[1]

??s
```

The ordering of the downvalues for `s` shows the degree of generality that can be determined, or else new rules are just added at the end of the list of downvalues.

```
DownValues[s]
```

```
Clear[s]
```

We now turn to some applications. Here is a function that has an arbitrary number of arguments (but at least three), in which the first and last arguments play a special role in the sense that they must definitively be present.

```
functionWithManyArguments[x_, y__, z_] := {{x}, {y}, {z}}
```

```
functionWithManyArguments[x, y, z]
```

First, the `Blank`s are matched and then the `BlankSequence`s are matched. For six arguments, we get the following result.

```
functionWithManyArguments[x, y1, y2, y3, y4, z]
```

For two arguments, `functionWithManyArguments` is not defined because __ (`BlankNullSequence[]`) assumes at least one argument.

```
functionWithManyArguments[x, z]
```

Patterns on the left-hand side of a definition with the same names only match identical arguments.

We should also note that `Pattern` has exactly two arguments: the name of the pattern and the pattern itself. Thus, the following construction, which tries to group by using bracketing to name a sequence of patterns, fails. It results in an expression with head `Pattern` and three arguments. No built-in rules exist for this construction.

```
a:Sequence[b_, c_]
```

The following construction also does not work. `Pattern` needs two arguments.

```
f[Pattern[a, b_, c_]] := {a, b, c}
```

The following definition, which also generates error messages, makes little sense. The pattern *x* is used for two different instances.

```
Clear[f]
f[x_, x__] := something
```

With _, __, and ___, it is possible to model significantly more complicated structures. Note that in the following example, the correspondence with `q` and `p` is determined by the `o`, which appears twice.

```
complFunc[o_, p_, q___, o_, s__] := {{o}, {p}, {q}, {s}}
```

```
complFunc[1, 2, 3, 4, 3, 2, 1, 5, 4, 5]
```

Warning: When using `BlankSequence` or `BlankNullSequence`, frequently several ways exist in which a pattern may be matched. *Mathematica* chooses the first one it finds.

Here is a function definition and a set of arguments in which more than one way exists to match the variables.

```
notUnique[a_, b__, c__, d_] := {{a}, {b}, {c}, {d}}
notUnique[a1, b1, b2, c1, c2, c3, d1]
```

But the matches `a`⟶`(a1)`, `b`⟶`(b1, b2)`, `c`⟶`(c1, c2, c3)`, and `d`⟶`(d1)` (and others) are also possible.

Warning: It is easy to get into an infinite loop using `BlankNullSequence`.

Here is such a situation. Because the right-hand side of the pattern matches the left-hand side without `y`, we get in an infinite loop.

```
p[x_, y___] := 2 p[x]
p[3]
```

We can see in detail how this happened using `Trace`. (To reduce the size of the output we use a smaller `$Recursion`·`Limit` value.)

```
oldRecursionLimitValue = $RecursionLimit;
$RecursionLimit = 20;
Trace[p[3]] // Short[#, 8]&

(* restore original value of $RecursionLimit *)
$RecursionLimit = oldRecursionLimitValue;
```

We now give a slightly more complicated example from physics using `BlankNullSequence`. In quantum field theoretical calculations, one frequently has to deal with expressions of the form [144✱]

$$\Gamma_n(d) = \sum_{\mu_1=1}^{d} \cdots \sum_{\mu_n=1}^{d} \gamma_{\mu_1}.\gamma_{\mu_2}.\cdots.\gamma_{\mu_n}.\gamma^{\mu_1}.\gamma^{\mu_2}.\ldots.\gamma^{\mu_n}$$

Here the γ_μ and γ^μ are noncommutative quantities (gamma matrices). They obey the following two simple rules

$$\gamma^\mu.\gamma_\nu + \gamma_\nu.\gamma^\mu = 2\,\delta^\mu_\nu\,\mathbf{1}_d$$

$$\sum_{\mu=1}^{d} \gamma^\mu.\gamma_\mu = d\,\mathbf{1}_d.$$

Here δ^μ_ν is the Kronecker symbol and $\mathbf{1}_d$ denotes the d-dimensional identity matrix.

$\Gamma_n(d)$ has the form $\Gamma_n(d) = f_n(d)\,\mathbf{1}_d$. We will calculate the function $f_n(d)$ for small positive integers n. We will write `γ[l[`*i*`]]` for γ_{μ_i} and `γ[u[`*i*`]]` for γ^{μ_i} (`l` and `u` standing for lower and upper). Later we use `𝓘[`*d*`]` for $\mathbf{1}_d$. Suppressing the implicitly understood summation, it is straightforward to implement the above rules (the third rule expresses the property of the Kronecker symbol). We use `a___`, `b___`, and `c___` to denote chains of γ_μ and/or γ^μ of unspecified length. The noncommutative multiplication we denote by p.

```
Clear[γ, p, l, u]

p[a___, γ[l[i_]], γ[u[j_]], b___] :=
       2 p[a, δ[i, j], b] - p[a, γ[u[j]], γ[l[i]], b]

p[a___, γ[l[i_]], γ[u[i_]], b___] := d p[a, b]

p[a___, γ[l[j_]], b___, δ[i_, j_], c___] := p[a, γ[l[i]], b, c]

p[] := 𝓘[d]
```

$\Gamma_2(d)$ is now easily calculated.

```
p[γ[l[1]], γ[l[2]], γ[u[1]], γ[u[2]]]
```

In the result for $\Gamma_3(d)$, one nicely sees how the rules were applied recursively.

```
p[γ[l[1]], γ[l[2]], γ[l[3]], γ[u[1]], γ[u[2]], γ[u[3]]]
```

Factoring the last output gives a much shorter result.

```
Factor[%]
```

Calculating $\Gamma_{10}(d)$ using the above rules takes about a minute and requires 353791 applications of the first, 843533 of the second, of the 671531 third, and 353792 of the fourth of the above definitions.

```
(γ10 = p[γ[l[1]], γ[l[2]], γ[l[3]], γ[l[4]], γ[l[5]],
         γ[l[6]], γ[l[7]], γ[l[8]], γ[l[9]], γ[l[10]],
         γ[u[1]], γ[u[2]], γ[u[3]], γ[u[4]], γ[u[5]],
         γ[u[6]], γ[u[7]], γ[u[8]], γ[u[9]], γ[u[10]]];) // Timing
```

`γ10` is a very large expression.

```
{LeafCount[γ10], ByteCount[γ10]}
```

After factorization, `γ10` becomes much more manageable.

```
Factor[γ10]
```

For $d = 4$, we could use the familiar form of the γ_μ and γ^μ to verify the last results. We will discuss the matrix operations that are used in the next inputs in the next chapter.

```
d = 4;
γ4[u[0]] = {{0, 0, 0, -I}, {0, 0, -I, 0}, {0, I, 0, 0}, {I, 0, 0, 0}};
γ4[u[1]] = {{0, 0, 0, -I}, {0, 0, I, 0}, {0, I, 0, 0}, {-I, 0, 0, 0}};
γ4[u[2]] = {{0, 0, 1, 0}, {0, 0, 0, -1}, {-1, 0, 0, 0}, {0, 1, 0, 0}};
γ4[u[3]] = {{I, 0, 0, 0}, {0, I, 0, 0}, {0, 0, -I, 0}, {0, 0, 0, -I}};

(* use metric with g[0, 0] == 1 *)
{γ4[l[0]],  γ4[l[1]],  γ4[l[2]],  γ4[l[3]]} =
{γ4[u[0]], -γ4[u[1]], -γ4[u[2]], -γ4[u[3]]};

(* check commutation relations *)
Table[γ4[u[i]].γ4[l[j]] + γ4[l[j]].γ4[u[i]] ==
      2 KroneckerDelta[i, j] IdentityMatrix[d],
      {i, 0, d - 1}, {j, 0, d - 1}]

(* check sum relation *)
Sum[γ4[u[i]].γ4[l[i]], {i, 0, 3}] == d IdentityMatrix[d]

(* carry out the direct summation *)
(Table[{n, Sum[Evaluate[Dot @@ Join[Table[γ4[l[μ[j]]], {j, n}],
                                    Table[γ4[u[μ[j]]], {j, n}]]],
               Evaluate[Sequence @@ Table[{μ[j], 0, d - 1}, {j, n}]]]},
      {n, 1, 8}]) ===
(* symbolic computation *)
(Table[{n, Factor[p @@ Join[Table[γ[l[μ[j]]], {j, n}],
                            Table[γ[u[μ[j]]], {j, n}]]]},
       {n, 1, 8}] /. I[d] -> IdentityMatrix[d])
```

In conjunction with the command `BlankNullSequence`, we now discuss again the function `Sequence`, introduced in Section 3.5. If `x` stands for several arguments in the following example, they must be extracted in one piece and enclosed in something. This something is `Sequence`.

```
Clear[f];
f[x___] := x
f[1, 2, 3, 4, 5]
```

An analogous but invisible application of `Sequence` takes place in this function definition (`Times` has the attribute `Flat` and so we have `Times[Sequence[1, 2, 3], Sequence[1, 2, 3]]` ⟶ $1\times2\times3\times1\times2\times3 = 36$).

```
Clear[times, x];
times[x___] := x*x;
times[1, 2, 3]
```

Using `Set` instead of `SetDelayed` in this example would have led to a different result, because `Sequence[1, 2, 3]` would have been substituted into `Power[x, 2]` as the first argument. As a result, `Power[1, 2, 3, 1, 2, 3]` evaluates to 1.

```
Clear[times, x];
times[x___] = x*x;
times[1, 2, 3]
```

We make one further remark concerning the use of the colon in named patterns. In the following simple case, the colon is superfluous.

```
Clear[f1, w];
f1[s_] := s^2;
f2[s:_] := s^2
{f1[w], f2[w]}
```

Next, we give several slightly more complicated constructions in which the colon (if it appears) plays a role. The difference in the various constructions is simply the amount of space between the letters inside the patterns and the use of the colon.

```
Clear[r, s, t];
♯1[s:_ t_]  := {s, t};
♯2[s:_t_]   := {s, t};
♯3[s_ t_]   := {s, t};
♯4[s_t _]   := {s, t};
♯5[s__:t]   := {s, t};
♯6[s _ t _] := {s, t};
```

Here are the results of these six functions for the argument `6r`.

```
{♯1[6r], ♯2[6r], ♯3[6r], ♯4[6r], ♯5[6r], ♯6[6r]}
```

We now examine these six cases in detail. To do this, we look at the full form and the input form of the various pattern expressions. (Using _, :, spaces, and symbols, many other patterns can be formed; we come back to this in the exercises at the end of this chapter.) In `♯1`, the pattern *s* is the product of something and the pattern *t*.

```
{InputForm[s:_ t_], FullForm[s:_ t_]}
```

Thus, the identification `s`⟶`6r`, `t`⟶`r` is possible.

```
♯1[6 r]
```

In `♯2`, *s* is the product of something and an object with the head `t`. In contrast to `♯1`, no space is between _ and `t`.

```
{InputForm[s:_t_], FullForm[s:_t_]}
```

Thus, `6r` does not fit the pattern, but for instance `56 t[45]` does.

```
♯2[6 r]
```

```
♯2[56 t[45]]
```

In the definition of `♯3`, the argument is the product of the pattern *s* and the pattern *t*, and so a correspondence with `6r` in the form `s`⟶`6`, `t`⟶`r` is possible.

```
{InputForm[s_ t_], FullForm[s_ t_]}
```

```
♯3[6 r]
```

♯4 is defined for arguments of the type *something* times the pattern *s* with head t.

```
{InputForm[s_t _], FullForm[s_t _]}
```

Here, the pattern is again not matched by 6r, but by 56 t[45].

```
♯4[6 r]
```

```
♯4[56 t[45]]
```

♯5 involves a structure we have not yet encountered; we discuss it at the beginning of the next subsection. Here, 6r fits the pattern via the correspondence s⟶6r and t⟶t.

```
{s__:t, FullForm[s__:t_]}
```

```
♯5[6 r]
```

The argument in ♯6 should have the structure of a product of s, t, and something squared.

```
{InputForm[s _ t _], FullForm[s _ t _]}
```

Thus, s t u^2 is, for instance, a suitable argument; but 6 r is not.

```
♯6[6 r]
```

```
♯6[s t u^2]
```

We now reexamine the command HoldPattern.

HoldPattern[*expression*]

is equivalent to *expression* as a pattern for pattern-matching purposes, but it does not evaluate *expression*.

This command is important if the pattern itself has to stay unevaluated, but we need to recognize it in its current form. Suppose, for example, that we want to define the function aPlusaPlusb for the argument a + a + b.

```
aPlusaPlusb[a_ + a_ + b_] := {a, a, b}
```

But using ??, we see that this result is not what we intended.

```
?? aPlusaPlusb
```

With HoldPattern, we can get what we want.

```
aPlusaPlusbHoldPatterned[HoldPattern[a_ + a_ + b_]] := {a, a, b}
```

```
?? aPlusaPlusbHoldPatterned
```

Still, applying this function fails in the next input because the argument is evaluated before testing the pattern (see the standard order for computations discussed in Chapter 4).

```
aPlusaPlusbHoldPatterned[a + a + b]
```

If we give this function the attribute HoldAll, this evaluation does not take place, and we get the desired result.

```
SetAttributes[aPlusaPlusbHeld, HoldAll];
aPlusaPlusbHeld[a_ + a_ + b_] := {a, a, b}
aPlusaPlusbHeld[a + a + b]
```

Without the attribute HoldAll, we can use Unevaluated to avoid the evaluation of the arguments.

```
aPlusaPlusbHoldPatterned[Unevaluated[a + a + b]]
```

Be aware that several functions behave differently when patterns are arguments. The following input with `Integrate` and a pattern argument does not evaluate to `Times[x_, y_]`.

```
Integrate[y_, x_]
```

```
Integrate[HoldPattern[y_], x_]
```

Other functions do not differentiate between patterns and non-patterns. (In a strict sense, `f[1]`, `f[x]` is a pattern; it can be used in definitions like `g[f[1]] :=` .... Here, we mean pattern in the sense of `Blank`... related.)

```
HoldPattern[x_] HoldPattern[x_]
```

```
D[HoldPattern[x_]^3, HoldPattern[x_]]
```

We saw the function `HoldPattern` in Chapter 3 when discussing downvalues. Left-hand sides of definitions are automatically wrapped in `HoldPattern`.

```
g[x + y] := x^y;
DownValues[g]
```

Inner occurrences of `HoldPattern` stay unchanged.

```
g[HoldPattern[x + y]] := x^y;
DownValues[g]
```

In the next chapters, we will need to use `HoldPattern` in patterns repeatedly. `HoldPattern` is an important function for writing large, rule-based programs. For efficiency, one often wants to avoid any evaluation in the left-hand sides of the rules. The following inputs list the standard packages that make use of `HoldPattern`.

```
files = Flatten[FileNames["*.m", #, Infinity]& /@
              Select[$Path, StringMatchQ[#, "*StandardPackages*"]&]];
```

```
Cases[Table[{files[[k]],
        Count[ReadList[Flatten[files][[k]], Hold[Expression]],
        HoldPattern, {-1}, Heads -> True]}, {k, Length[files]}],
        {_, _?(# =!= 0&)}]
```

Because left-hand sides of definitions are wrapped in `HoldPattern`, the function `HoldPattern` is a very frequently encountered function in *Mathematica* calculations. The following input counts the number of times the function `HoldPattern` is encountered when evaluating the integral $\int \sin(x^3)\, dx$.

```
((* keep where messages are sent to and processing function *)
 old$Messages = $Messages;
 old$MessagePrePrint = $MessagePrePrint;
 (* a bag for collecting the steps *)
 bag = {};
 (* as a side effect, collect all steps *)
 $MessagePrePrint = ((bag = {#, bag})&);
 (* redirect messages *)
 $Messages = nowwhere;
 On[];
 (* do the integration *)
 Integrate[Sin[x^3], x];
 Off[];
 (* restore where messages are sent to and old preprocessing *)
 $Messages = old$Messages;
 $MessagePrePrint = old$MessagePrePrint;
 Count[bag, HoldPattern, {-1}, Heads -> True]) // Timing
```

Sometimes we need to match patterns literally, for instance, when writing programs that autogenerate programs, which can be done with the function `Verbatim`.

`Verbatim[`*pattern*`]`

is used to match *expression* as a pattern.

The following function `matchThePattern` has a special definition for the pattern “`x_`” itself.

```
matchThePattern[Verbatim[x_]] := thePatternItselfWasThere
```

```
matchThePattern[x_] := someOtherArgumentWasThere
```

According to the general rule, special definitions come first.

```
?matchThePattern
```

The `Verbatim` pattern matches only if the argument of `matchPattern` is `x_`.

```
matchThePattern[__]
```

```
matchThePattern[y_]
```

```
matchThePattern[x_]
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

■ 5.2.2 Patterns with Special Properties

Frequently, we want to be able to change certain parameters in a function, but we do not want to have to write out all of the parameters explicitly. One possibility would be to use the command `Options` discussed in Chapter 3, but it is somewhat unusual to use it in relation to “parameters” and requires more typing than needed. Another possibility is `Optional`.

`Optional[`*pattern*`, `*default*`]`

or

pattern`:`*default*

represents a pattern *pattern* that may not appear explicitly, in which case, the default *default* is used.

Note the order of `_` and `:` in the following expressions. The interesting structure here is `x_:y`.

```
{FullForm[x_:y], FullForm[x:_y], FullForm[x:y], FullForm[x_:_y]}
```

Here is a definition of a function with optional argument `y`.

```
defaultFunc[x_, y_:yDefault] := x + y
```

If `y` is given explicitly, it is used.

```
defaultFunc[ξ, η]
```

If not, the default value is used.

```
defaultFunc[ξ]
```

In the next example, we use the colon `:` twice, one time as the shorthand for `Pattern` and one time as the shorthand for `Optional`.

```
Clear[f, x, y]
```

```
f[y:(x_):1] := {x, y}
```

Using `FullForm`, we see the double meaning.

```
x:(x_):1 // FullForm
```

Note that brackets were needed in the last input.

```
y:x_:1 // FullForm
```

```
y:(x_:1) // FullForm
```

The two pattern variables `x` and `y` represent the same pattern, so we have the following example (here, we make use of the optionality of the argument).

```
f[]
```

If an argument is explicitly given, it is used.

```
f[3]
```

A possible choice for optional arguments is `Automatic`. `Automatic` is also often used in possible option values. Using `Automatic` makes it possible to distinguish among various cases in a natural way. Here is an example of a function `optFunc` with two optional arguments, both of which have the default value `Automatic`.

```
optFunc[x_, o1_:Automatic, o2_:Automatic] :=
If[o1 === Automatic,
    If[o2 === Automatic, {x, Automatic, Automatic},
                         {x, Automatic, notAutomatic}],
    If[o2 === Automatic, {x, notAutomatic, Automatic},
                         {x, notAutomatic, notAutomatic}]]
```

Here is what we get with various second and third arguments, or without them.

```
optFunc[1, Automatic, Automatic]
```

```
optFunc[1, 2, Automatic]
```

```
optFunc[1, Automatic, 3]
```

```
optFunc[1]
```

```
optFunc[2]
```

```
optFunc[1, 2]
```

```
optFunc[1, 3]
```

Be aware that the *pattern* in `Optional` cannot be an arbitrary complex pattern; in most cases, *var*_ is used. Optional values must match the corresponding pattern. Here, the pattern describes an integer, but the optional value is real, so it does not work properly.

```
doesNotWork[pattVar_Integer: -2.5] = pattVar
```

```
doesNotWork[-3.4]
```

```
doesNotWork[4]
```

```
doesNotWork[]
```

Here, it works well.

```
doesWork[pattVar_Integer: -5] = pattVar;
```

```
doesWork[5]
```

```
doesWork[]
```

Without any type restrictions on the pattern, we, of course, always get a nontrivial result.

```
doesAlsoWork[pattVar_: -2.5] = pattVar
```

```
doesAlsoWork[-3]
```

```
doesAlsoWork[]
```

It is also possible to assign an optional value to a function for certain arguments outside of the function definition (using `Default`; we come back to this function later in the chapter). Then, the structure simplifies to `Optional[`*x*`_]` or *x*`_.` (the period belongs to the *Mathematica* expression).

`Optional[`*pattern*`]`
or
pattern`_.`
represents a pattern *pattern* that may not appear explicitly, in which case, the previously defined default value is used.

Among the system functions, `Plus`, `Times`, and `Power` have such predefined optional arguments.

`Plus`, `Times`, and `Power` have internally defined optional values:

`x_ + y_.`
default for `y_.` is 0

`x_ y_.`
default for `y_.` is 1

`x_^y_.`
default for `y_.` is 1

Thus, `x` becomes `x + 0` with `Plus`, and `x` becomes `x*1` with `Times`.

```
{Plus[x], Times[x]}
```

The following function `defaultTest` makes use of all three of the default possibilities shown above.

```
Remove[a, b, c, x, defaultTest];
defaultTest[(a_. + b_. x)^c_.] := {a, b, c, x}
```

For an arbitrary argument, `defaultTest` gives the expected result.

```
defaultTest[(12 x + 34)^w]
```

In the following case, `defaultTest` uses the default values.

```
defaultTest[only x]
```

```
defaultTest[onlyC + x]
```

```
defaultTest[(1 + x)^2]
```

```
defaultTest[x]
```

But using a symbol other than `x` does (of course) not match the pattern.

```
defaultTest[a + b y]
```

It sometimes happens that the arguments in functions repeat (and in defining the function, we know how often they appear). For such cases, an appropriate pattern is `Repeated`.

`Repeated[`*pattern*`]`
or
pattern`..`
represents one or more appearances of the pattern *pattern*.

`RepeatedNull[`*pattern*`]`
or
pattern`...`
represents zero or more appearances of the pattern *pattern*.

The following function `repeat` gives the repeated variable and the number of times it appears. Note the braces around `b` because of the `Sequence` enclosing it. The pattern *b* matches more than one argument.

```
repeat[b:((a:_)..)] := {a, Length[{b}]}
```

Here is the `FullForm` of the inside expression.

```
FullForm[b:((a:_)..)]
```

This function does the expected.

```
repeat[a, a, a, a]
```

```
repeat[{ϒ, ϒ}, {ϒ, ϒ}, {ϒ, ϒ}, {ϒ, ϒ}, {ϒ, ϒ}]
```

In the following call on `repeat`, the pattern is not matched because only the repeated pattern can appear.

```
repeat[a, a, a, a, 1]
```

When called with no arguments the current definition for `repeat` does not match.

```
repeat[]
```

The following function is defined to accept the previous input where the last argument was different.

```
repeat2[b:((a:_)..), x_] := {{a, Length[{b}]}, x}
repeat2[a, a, a, a, 𝕒]
```

When several ways exist to match the patterns, the blanks (head `Blank`) are first matched (if possible), as usual.

```
repeat3[b:((a:_)..), x__] := {{a, Length[{b}]}, x}
repeat3[a, a, a, a, 𝕒]
```

Using `...` instead of `..` makes the definition match the zero-argument case.

```
repeat4[b:((a:_)...)] := zeroArguments
repeat4[]
```

Sometimes it is convenient to specify a pattern that should not be matched (instead of specifying all patterns that should be matched). This can be done with the function `Except`.

`Except[`*pattern*`]`
represents a pattern that matches anything with the exception of *pattern*.

Here is a function that is defined for any argument other than expressions with the head `Real`.

```
notDefinedForReals[x:Except[_Real]] := x^2
```

The function evaluates for integers, symbols, complex numbers, but not real numbers.

```
{notDefinedForReals[2], notDefinedForReals[2 + 2 I],
 notDefinedForReals[αβγ],
 notDefinedForReals[3.14]}
```

Often, we want to define functions under very restrictive conditions, much more restrictive than simply matching head specifications. In principle, this is possible by using `If[...]` in the corresponding function definition. However, a faster and more elegant and understandable approach is to test the pattern itself (on the left-hand side of the function definition) to see which possible definition to use. A first extension in this direction is to allow the possibility of several patterns.

`Alternatives[`$pattern_1$`, `$pattern_2$`, ..., `$pattern_n$`]`
or
$pattern_1$ | $pattern_2$ | ⋯ | $pattern_n$
represents the various possibilities $pattern_i$ of a pattern.

The following function `ORA` (shortcut for only real arguments) is defined only for real-valued arguments; that is, the head of the argument must be `Integer`, `Rational`, or `Real`. It is not defined for complex or symbolic arguments.

```
ORA[x_Integer | x_Rational | x_Real] := x
```

For complex or symbolic arguments, it remains unevaluated.

```
{ORA[1], ORA[2.6], ORA[56/67], ORA[1 + 10^-23 I], ORA[𝕍], ORA["abc"]}
```

The following notation is also possible.

```
ORB[x:(_Integer | _Rational | _Real)] := x
```

```
{ORB[1], ORB[2.6], ORB[56/67], ORB[1 + 10^-23 I], ORB[𝕍], ORB["abc"]}
```

But not this notation.

```
ORC[x:(_(Integer | Rational | Real))] := x
```

```
{ORC[1], ORC[2.6], ORC[56/67], ORC[1 + 10^-23 I], ORC[𝕍], ORC["abc"]}
```

The pattern test for the head in `ORA` refers to the "real" head of the variables, so the following two arguments do not match. Despite `iAmReallyAnIntegerBelieveMe`'s attempts to hide its real nature the following does not work.

```
iAmReallyAnIntegerBelieveMe/:
              IntegerQ[iAmReallyAnIntegerBelieveMe] = True
ORA[iAmReallyAnIntegerBelieveMe]
```

```
iAmReallyAnIntegerBelieveMe/:
              Head[iAmReallyAnIntegerBelieveMe] = Integer
ORA[iAmReallyAnIntegerBelieveMe]
```

In this example, a very special type of argument is required, namely, the product of something with the sin or cos of something.

```
Clear[g, t, r];
g[a_ (b:(Sin | Cos))[x_]] := {a, b, x}
```

Here, `13 Cos[t^2 + r]` has this form.

```
g[13 Cos[t^2 + r]]
```

However, `13 soC[t^2 + r]` does not.

```
g[13 soC[t^2 + r]]
```

To test values of the arguments as well as patterns, we can use PatternTest.

PatternTest[*pattern*, *test*]
or
pattern?*test*

represents the pattern *pattern* if the test *test* is applied to the actual argument evaluates to True. Here, *test* must be a (pure) function.

The following function is defined only for rational arguments larger than 45/91.

```
rationalOnly[x_Rational?((# > 45/91)&)] := x

{rationalOnly[45/91], rationalOnly[46/91],
 rationalOnly[5], rationalOnly[tgh]}
```

Be sure to note the use of parentheses after the ?. The pure function's & binds very weakly.

```
FullForm[x_?f[#]&]
```

Here is what we wanted.

```
FullForm[x_?(f[#]&)]
```

This input is shorter.

```
FullForm[x_?f]
```

A previously defined function can also be used in PatternTest.

```
ℒ[x_] := If[x > 3, True, False]

𝔣1[x_?ℒ] := {x}

𝔣2[x_?(ℒ[#]&)] := {x}

{𝔣1[2], 𝔣1[4], 𝔣2[2], 𝔣2[4]}
```

Note that the symbols inside of PatternTests lie below level 2 of the expression, and rules cannot be attached to them. So the following attempt to set up a rule for x, which should fire whenever x appears somewhere in an expression, fails.

```
Clear[x, y]

x /: y_?(MemberQ[#, x, {0, Infinity}, Heads -> True]&) := Print[y]

y_?(MemberQ[#, x, {0, Infinity}, Heads -> True]&) // TreeForm

Position[y_?(MemberQ[#, x, {0, Infinity}, Heads -> True]&), x]
```

Much more complicated structures can be built using these various patterns and tests. In the following complicated⋱ Function, the first argument must be an even number, the second a product, the third real-valued or rational, the fourth an integer, the fifth must be present, and at least one or more arguments with head List must follow.

```
complicatedFunction[(u_)?(EvenQ[#]&), v_Times,
                    w_Real | w_Rational,
                    x_Integer, y__, z__List] :=
(Print["u = ", {u}]; Print["v = ", {v}]; Print["w = ", {w}];
 Print["x = ", {x}]; Print["y = ", {y}]; Print["z = ", {z}])
```

Here, we apply it to some arguments. Note that the last `Sequence` disappears before the pattern matching process.

```
Clear[e, r];
complicatedFunction[4, e r, N[Pi], 12321223, 2, e r,
                    {8, 9}, 3, {2, 1}, {Null, r}, {4}, Sequence[]]
```

In the following example, the pattern is not matched because the first argument is not an even number. (This time we do not explicitly write the `Null` element.)

```
complicatedFunction[5, e r, N[Pi], 12321223, 2, e r,
                    {8, 9}, 3, {2, 1}, {, r}, {4}]
```

Using `PatternTest` together with `BlankSequence` and `BlankNullSequence` can sometimes lead to misunderstandings. For example, consider the following definition.

```
Remove[f]
f[x__?((Length[{#}] > 1)&)] := {x, y}

f[1, 2, 3]
```

It does not produce the "expected" `{1, 2, 3}`. To see why, we include `Print` in the `PatternTest`.

```
Remove[f]
f[x__?((Print[{#}]; Length[{#}] > 1)&)] := {x, y}

f[1, 2, 3]
```

Thus, the test is performed for every argument, and because it fails on the first argument, the evaluation stopped, because it is now clear that the pattern does not match. Sometimes it is difficult, and even impossible, to restrict the applicability of definitions and patterns using `PatternTest`, especially if multiple arguments of a function have to fulfill some cross-relations. As an alternative to `PatternTest`, we have `Condition`.

`Condition[`*expression*`, `*condition*`]`
or
expression `/;` *condition*
restricts the applicability of *expression* to the cases in which *condition* is `True`.

The big advantage of `Condition` compared with `PatternTest` is that it allows the use of named variables. `Condition` can be used for patterns as well as for *Mathematica* expressions.

`Condition` can be used in conjunction with `Pattern`, `SetDelayed`, `RuleDelayed`, `Block`, `With`, and `Module`. For the sake of efficiency and understandability, `Condition` should be used in `Pattern` if possible, and not after the entire expression.

Here are two ways to specify the same restriction that do exactly the same thing, although the first is preferred. Here is the first possibility.

```
cond1[x_ /; 1 < x < 2] := x
```

Here, grouping is used to specify the restriction.

```
FullForm[x_ /; 1 < x < 2]
```

The second possibility is to put `Condition` on the right-hand side of the `SetDelayed` definition.

```
cond2[x_] := x /; 1 < x < 2
```

Here, we write it out.

```
FullForm[Hold[cond2[x_] := x /; 1 < x < 2]]
```

`cond1` and `cond2` code the same patterns.

```
{cond1[0], cond1[3/2], cond2[0], cond2[3/2]}
```

Note that in this case we also could have used a pure function inside the pattern using `PatternTest`: `cond[x_?(1<#<2&)]:=x`.

A `Condition` *condition* can also be given "in one piece" on the left-hand side of a definition instead of inside the pattern on the right-hand side (as in the example before the last one), or on the right-hand side of an assignment (as in the last example).

```
(cond3[x_, y_] /; x < y) = {x, y}
```

We write this expression out again to better identify the structure.

```
FullForm[Hold[f[x_, y_] /; x < y = {x, y}]]
```

This construction also works as expected.

```
{cond3[1, 2], cond3[2, 1]}
```

Inside the *condition* appearing in `Condition[`*pattern*`,` *condition*`]`, we can also test variables that are not pattern variables. As a side effect in the pattern test in the function `cond4`, we change the value of `a`.

```
Clear[cond4, a, x];
a = 0;
cond4[x_ /; (a = a + 1; a > 2)] := {a, x}
```

```
??cond4
```

Reevaluating `cond4` five times, give different results.

```
Do[Print[a, "  ", cond4[i]], {i, 1, 5}]
```

`a` has now the value 5.

```
a
```

One can also have a compound expression on the right-hand side of a set or set delayed definition that ends with a condition head `Condition`). In this case, the first elements of the compound expression are evaluated, but the whole function returns unevaluated. Here is an example of this situation.

```
fABC[x_] := ((setA = 1); (setB = 2); (setC = 3) /; False)
```

```
Hold[fABC[x_] := ((setA = 1); (setB = 2); (setC = 3) /; False)] // FullForm
```

```
fABC[1]
```

```
{setA, setB, setC}
```

As stated earlier, one of the big advantages of `Condition` is that the variable names themselves can be used, which allows relationships between variables to be used as restrictions on the definition of a function, outside of the pattern. Here, the use of `Condition` is much more difficult to avoid.

```
Clear[f];
f[x_, y_] := {x, y} /; x > y
{f[1, 2], f[2, 1]}
```

The following example also works.

```
Clear[f];
(f[x_, y_] /; x > y) := {x, y}
{f[1, 2], f[2, 1]}
```

So does this example.

```
Clear[f];
f[x_, y_] := ({x, y} /; x > y)
{f[1, 2], f[2, 1]}
```

But this example does not work, because it is syntactically not allowed.

```
Clear[f];
f[(x_, y_) /; x > y] := {x, y}

{f[1, 2], f[2, 1]}
```

Inside `Module`, local variables can be used in `Condition` as well as in expressions. Here, the function `f[x]` is defined only under certain conditions; whether these conditions are satisfied is tested inside `Module`. Note that if the test carried out by `Condition` is not satisfied inside `Module`, the function remains unevaluated. In the following example nothing is printed.

```
Clear[f];
f[x_] := Module[{y = x}, (Print[{x, y}]; y^2) /; y^3 < 0]

f[1]
```

Now, the condition is fulfilled.

```
f[-1]
```

The local variable must have a value from the beginning for this construction to work.

```
Clear[f];
f[x_] := Module[{y}, (y = x; Print[{x, y}]; y^2) /; y^3 < 0]

f[-1]
```

And the `Condition` must be literally present in the beginning of the evaluation process.

```
Clear[f];
f[x_] := Module[{condition = Condition, y = x},
                condition[y = x; Print[{x, y}]; y^2, y^3 < 0]]

f[-1]
```

Analogous constructions are also possible with `Block` and `With`. Such constructions are very valuable when the test of the applicability of a rule is very expensive. The calculations carried out in the test have a large overlap with the calculations needed for generating the result. Here is an example.

```
Clear[D];
D[trueFalse_] :=
 Module[{testAndResult = resultOfALongCalculation[res, trueFalse]},
        testAndResult[[1]] /; testAndResult[[2]]]

D[True]

D[False]
```

Note the different behavior of `PatternTest` and `Condition` when used with `BlankSequence` and `BlankNull`-`Sequence`. `PatternTest` tests each element individually. Here is a definition for a function `f` that never applies.

```
Clear[f];
f[x__?((Print[#]; False)&)] := {x}
```

```
FullForm[x__?((Print[Hold[#]]; False)&)]
```

Next, we call `f` with four arguments. The first element is tested by itself, and because the result is `False`, no further tests are carried out.

```
f[1, 2, 3, 4]
```

Here is an analogous construction with `Condition`. Now, `x` is replaced by the combination of all arguments.

```
Clear[f, g];
f[x__ /; (Print[Unevaluated[x]]; False)] := {x}

FullForm[x__ /; (Print[Unevaluated[x]]; False)]

f[1, 2, 3, 4]
```

Here is one more similar example.

```
g[a___ /; Length[{a}] > 2] := {a}

{g[1, 2], g[1, 2, 3]}
```

`PatternTest` and `Condition` are two general constructs to restrict patterns. Because they carry out a larger amount of work than simply checking a type using `Blank[`*type*`]`, carrying them out needs more time. The next input compares three possibilities to restrict an argument to be an integer. Clearly, the first is the fastest and shortest.

```
𝕗1[k_Integer] = k;
𝕗2[k_?(Head[#] === Integer&)] = k;
𝕗3[k_ /; Head[k] === Integer] = k;

Timing[Do[𝕗1[k], {k, 10^5}]]

Timing[Do[𝕗2[k], {k, 10^5}]]

Timing[Do[𝕗3[k], {k, 10^5}]]
```

Using `PatternTest` and `Condition`, it is possible to program very specific patterns. Consider the game *Sorry* with a "typical" die with one player. (The case of several players without elimination is trivial, whereas the case with elimination can be recursively programmed in a similar way.) Then, we find the number of possible configurations (without expropriation) in one game with m players and n squares (this is just the content of the following definition of ϕ). (`ϕ[`*m*`][`*n*`]` represents the number of different ways for $\phi_m(n)$ to represent the positive integer n as a sum of positive integers $< m$ taking into account order.)

```
ϕ[m_][n_?(# < 0&)] = 0;
ϕ[1][n_] = 1;
ϕ[m_][n_] := (ϕ[m][n] = ϕ[m][m]) /; m > n
ϕ[m_][m_] := ϕ[m][m] = 1 + ϕ[m - 1][m];
ϕ[m_][n_] := (ϕ[m][n] = 2 ϕ[m][n - m] +
              Sum[ϕ[m][i] ϕ[m][m] ϕ[m][n - m - i], {i, n - m - 1}]) /; n >

ϕ[6][46]

N[%]
```

Note that `Condition` is sometimes only used on the right-hand side in `SetDelayed` and `RuleDelayed`. The following is in most cases not the wanted definition of `f`.

```
Clear[f, x];
f[x_] = x^2 /; x > 0;
f[1]
```

The possibility of specifying well-defined patterns that are applied only in relevant situations is very important for building and using complicated sets of rules (e.g., equations, definitions). A practical, more useful, and a bit more complicated example is the calculation of

$$\int_0^{\pi} \frac{\cos^c(\vartheta)\sin^s(\vartheta)}{\left(1-k^2\sin^2(\vartheta)\right)^n\sqrt{1-k^2\sin^2(\vartheta)}}\,d\vartheta$$

in complete elliptic integrals for nonnegative integers c, s, and n and $0 \leq k < 1$ ([93✱]).

Let `SC[n, s, c, k]` be the above integral. For odd c, the integral is always zero by symmetry of the integrand around $\vartheta = \pi/2$.

The following recursive relations exist for the boundaries of the n,s,c-parameter space. The special conditions of their applicability are encoded in the appropriate `PatternTest` on the left-hand side of the definitions. (We do not prove them here, but just use them; see the cited reference for details.)

```
(* clear all variable to be used *)
Clear[SC, n, s, c, k, l, writeNicely, myIntegrate]

SC[n_Integer?(# >= 0&), s_Integer?(# >= 0&), c_Integer?OddQ, k_] = 0;

SC[0, s_Integer?(# >= 4&), 0, k_] :=
  ((s - 2)(1 + k^2))/((s - 1) k^2) SC[0, s - 2, 0, k] -
   (s - 3)/((s - 1) k^2) SC[0, s - 4, 0, k]

SC[0, 0, c_Integer?(# >= 4 && EvenQ[#]&), k_] :=
  ((c - 2)(2k^2 - 1))/((c - 1) k^2) SC[0, 0, c - 2, k] -
   ((c - 3)( k^2 - 1))/((c - 1) k^2) SC[0, 0, c - 4, k]

SC[n_Integer?(# >= 0&), s_Integer?(# >= 2&), 0, k_] :=
  1/k^2(SC[n, s - 2, 0, k] - SC[n - 1, s - 2, 0, k])

SC[n_Integer?(# >= 0&), 0, c_Integer?(# >= 2 && EvenQ[#]&), k_] :=
  1/k^2(SC[n - 1, 0, c - 2, k] - (1 - k^2)SC[n, 0, c - 2, k])

SC[0, s_Integer?(# >= 4&), 2, k_] := (
  (s + (s - 2)k^2)/((s + 1)k^2) SC[0, s - 2, 2, k] -
  (s - 3)/((s + 1) k^2) SC[0, s - 4, 2, k])

SC[0, s_Integer?(# >= 0&), c_Integer?(# >= 4 && EvenQ[#]&), k_] :=
  (((s + c - 2)(2k^2 - 1) - s k^2)/((s + c - 1) k^2)SC[0, s, c - 2, k] +
   ((c - 3)(1 - k^2))/((s + c - 1) k^2) SC[0, s, c - 4, k])

SC[1, 1, c_Integer?(# >= 4 && EvenQ[#]&), k_] :=
  ((c - 1)(2k^2 - 1) - 3k^2)/((c - 2)k^2) SC[1, 1, c - 2, k] +
   ((c - 3)(1 - k^2))/((c - 2) k^2) SC[1, 1, c - 4, k]
```

The following two relations are more general and apply to the inner points of the n,s,c-space.

```
SC[n_Integer?(# >= 1&), s_Integer?(# >= 2&),
   c_Integer?(# >= 2 && EvenQ[#]&), k_] :=
1/k^2(SC[n, s - 2, c, k] - SC[n - 1, s - 2, c, k])

SC[n_Integer?(# >= 2&), s_Integer?(# >= 0&),
   c_?(# >= 0 && EvenQ[#]&), k_] :=
(s - c - (2 - k^2)(s - 2n + 2))/((2n - 1)(1 - k^2)) SC[n - 1, s, c, k] +
(s + c - 2n + 3)/((2n - 1)(1 - k^2)) SC[n - 2, s, c, k]
```

(If we knew that we would have to calculate a lot of integrals of the type above, a `SetDelayed[SC[n_, s_, c_, k_], Set[SC[n_, s_, c_, k_], …]]` construction would be more appropriate because this would allow us to remember the already-calculated values.)

These recursive relations have to be supplemented by starting values near the {0, 0, 0} corner of the *n*, *s*, *c*-lattice (`EllipticE` and `EllipticK` are complete elliptic integrals, which we discuss in Chapter 3 of the Symbolics volume [140✱]).

```
SC[0, 0, 0, k_] = 2 EllipticK[k^2];
SC[1, 0, 0, k_] = 2 EllipticE[k^2]/(1 - k^2);
SC[0, 0, 2, k_] = 2/k^2 (EllipticE[k^2] + (k^2 - 1) EllipticK[k^2]);
SC[0, 2, 0, k_] = 2/k^2 (EllipticK[k^2] - EllipticE[k^2]);
SC[0, 2, 2, k_] = 2/3 ((2 - k^2)/k^4 EllipticE[k^2] +
                   2(k^2 - 1)/k^4 EllipticK[k^2]);
SC[0, 3, 2, k_] = 1/(8 k^4) (2(3 - k^2) -
                   (3 + k^2)(1 - k^2) SC[0, 1, 0, k]);
SC[0, 1, 0, k_] = 1/k Log[(1 + k)/(1 - k)];
SC[1, 1, 0, k_] = 2/(1 - k^2);
SC[0, 1, 2, k_] = 1/(2k^2) (2 - (1 - k^2)SC[0, 1, 0, k]);
SC[1, 1, 2, k_] = 1/(k^2)  (SC[0, 1, 0, k] - 2);
SC[0, 3, 0, k_] = 1/(2k^2)((1 + k^2)SC[0, 1, 0, k] - 2);
```

Let us look at what we have implemented.

```
FullDefinition[SC]
```

We see that *Mathematica* has reordered the rules to apply the special ones before the more general ones.

We further define a function `writeNicely` simplifying the large expressions from the recursive calculation. (This function uses some commands we discuss in Chapter 1 of the Symbolics volume [140✱].) The function `writeNicely` writes the expression as a sum of prefactors times elliptic integrals or logarithms. In addition, it transforms expressions of the form `Sqrt[`*k*`^2]` to *k* because of the given above restrictions, which apply for *k*.

```
writeNicely[expr_, h_] :=
Module[{mainTerms, collected, elTerms, rest},
       expr1 = PowerExpand[expr, Level[h, {-1}]];
       (* select elliptic functions *)
       mainTerms = Cases[expr1, _EllipticE | _EllipticK | _Log,
                         {0, Infinity}] // Union;
       collected = Collect[expr1, mainTerms];
       (* write as sum of summands of the form
    rational * elliptic function *)
       elTerms = Cases[collected, _ _EllipticE | _ _EllipticK | _ _Log];
       (* factor the rational part *)
       (rest = Total[Factor /@ elTerms]) + Factor[Expand[expr1 - rest]]]
```

So, we can finally define a function `myIntegrate` calculating these integrals here and writing them in an appropriate form. (We could, of course, also use `Unprotect` the function `Integrate` and associate this rule with the built-in command `Integrate`.) We also use the pattern `(1 + h_ Sin[t_]^2)^v_` to match numeric quantities, which would not have the structure `Plus[1, Times[-1,` *number*`, Power[Sin[t], 2]]]`, but rather `Plus[1, Times[−`*number*`, Power[Sin[t], 2]]]` and not only symbolic values for `h`.

```
myIntegrate[Sin[t_]^s_. Cos[t_]^c_. *
            (1 + h_ Sin[t_]^2)^v_, {t_, 0, Pi}] :=
If[Evaluate[0 <= -h < 1], Evaluate[
   writeNicely[SC[-v - 1/2, s, c, Sqrt[-h]], h]], hereNotDone] /;
    (IntegerQ[-v - 1/2] && - v - 1/2 >= 0 s >= 0 && c >= 0 &&
     Head[s] == Integer && Head[c] == Integer)

myIntegrate[Sin[t_]^s_.(1 + h_ Sin[t_]^2)^v_, {t_, 0, Pi}] :=
If[Evaluate[0 <= -h < 1], Evaluate[
   writeNicely[SC[-v - 1/2, s, 0, Sqrt[-h]], h]], hereNotDone] /;
   (IntegerQ[-v - 1/2] && - v - 1/2 >= 0 s >= 0 && Head[s] == Integer)

myIntegrate[Cos[t_]^c_. (1 + h_ Sin[t_]^2)^v_, {t_, 0, Pi}] :=
If[Evaluate[0 <= -h < 1], Evaluate[
   writeNicely[SC[-v - 1/2, 0, c, Sqrt[-h]], h]], hereNotDone] /;
   (IntegerQ[-v - 1/2] && - v - 1/2 >= 0 && c >= 0 && Head[c] == Integer)

myIntegrate[(1 + h_ Sin[t_]^2)^v_, {t_, 0, Pi}] :=
If[Evaluate[0 <= -h < 1], Evaluate[
   writeNicely[SC[-v - 1/2, 0, 0, Sqrt[-h]], h]],
   hereNotDone] /; (IntegerQ[-v - 1/2] && - v - 1/2 >= 0)
```

Let us try some examples.

```
myIntegrate[Sin[t]^4 Cos[t]^6/(1 - k Sin[t]^2)^(5/2), {t, 0, Pi}]
```

This result agrees with the result of the built-in function `Integrate`.

```
(Integrate[Sin[t]^4 Cos[t]^6/(1 - k Sin[t]^2)^(5/2),
            {t, 0, Pi}, Assumptions -> 0 < k < 1] /.
   (* use only EllipticK[k] and EllipticE[k] *)
   {EllipticE[k/(k - 1)] -> EllipticE[k]/Sqrt[1 - k],
    EllipticK[k/(k - 1)] -> Sqrt[1 - k] EllipticK[k]} // Simplify) /.
   (* factor prefactors *) p_Plus?(PolynomialQ[#, k]&) :> Factor[p]

(* use indefinite integral and substitute limits *)
Collect[(# /. t -> Pi) - (# /. t -> 0),
        (* write as sum of two elliptic integrals *)
        _EllipticK | _EllipticE, Factor]& @
        Integrate[Sin[t]^4 Cos[t]^6/(1 - k Sin[t]^2)^(5/2), t]
```

Here are some more examples.

```
myIntegrate[Sin[s]^6 Cos[s]^4/(1 - k Sin[s]^2)^(3/2),
            {s, 0, Pi}]

myIntegrate[Sin[s]^6 Cos[s]^5/(1 - k Sin[s]^2)^(3/2),
            {s, 0, Pi}]
```

For some parameters, the integral contains only `Log` functions and no elliptic integrals.

```
myIntegrate[Sin[t]^7/(1 - l^2 Sin[t]^2)^(5/2), {t, 0, Pi}]
```

Sometimes, even `Log` can be absent.

```
myIntegrate[Sin[t]^3 Cos[t]^4/(1 - k^2 Sin[t]^2)^(7 + 1/2),
            {t, 0, Pi}]
```

For more examples of integrals that can be carried out recursively, see [10✱].

Patterns frequently allow for many possible realizations. In the next input, there are two possible realizations for the patterns `A` and `B`. *Mathematica* chooses the “second” one (although the expression as well as the pattern are already in canonical order).

```
(a1 + a2 a3) (b1 + b2 b3) /.
      (A:(α_)) (B:(β1_ + β2_)) :> {"A"⟹ A, "B" ⟹ B}
```

As a rule of thumb, patterns with more complicated (deeply specified) subpatterns are matched before simple patterns are matched and patterns with `Blank[]` are matched before patterns with `BlankSequence[]` and `BlankNullSe⋱ quence[]` are matched. Here are three examples.

```
(a1 + a2 a3) (b1 + b2 b3) (c1 + c2 c3) /.
    (* three equal simple patterns *)
   (A:(_)) (B:(_)) (Γ:(_)) :> {"A"⟹ A, "B" ⟹ B, "Γ" ⟹ Γ}
```

```
(a1 + a2 a3) (b1 + b2 b3) (c1 + c2 c3) /.
    (* three increasingly complicated patterns *)
   (A:(α_)) (B:(β1_ + β2_)) (Γ:(γ1_ + γ2_ γ3_)) :>
                          {"A"⟹ A, "B" ⟹ B, "Γ" ⟹ Γ}
```

```
(a1 + a2 a3) (b1 + b2 b3) (c1 + c2 c3) /.
    (* three increasingly less restrictive patterns *)
   (A:(α_)) (B:(β__)) (Γ:(γ___)) :>
                          {"A"⟹ A, "B" ⟹ B, "Γ" ⟹ Γ}
```

n the next pattern that includes pattern tests, the three factors are matched in their original order.

```
(a1 + a2 a3) (b1 + b2 b3) (c1 + c2 c3) /.
   (A:(_?(True&))) (B:(_?(True&))) (Γ:(_?(True&))) :>
                            {"A"⟹ A, "B" ⟹ B, "Γ" ⟹ Γ}
```

In general, one should not rely on a certain chosen pattern match, which might be *Mathematica* version specific. In ambiguous cases (meaning multiple possible matches exist) where the matches matter, it is always best to use additional pattern tests to force a unique pattern match.

We make a short remark about possible exceptional situations. As discussed in the last chapter, the evaluation process of a *Mathematica* expression proceeds recursively until no changes occur anymore. While correct as an idealized theoretical concept, in practice various shortcuts are in place to speed up the infinite recursive evaluation. As a result, in some situation the recursive evaluation is not carried out as far as it should. These rare situations happen frequently when `Condition` is used.

Here is a definition for the value `𝕓[β]` that applies when the value of `𝕓Equal1Q` is `True`.

```
𝕓[β] := 1 /; 𝕓Equal1Q
```

In the next inputs, the definition does not apply. We store the value of `𝕓[β]` using the symbol `iEvaluateLazily`.

```
𝕓Equal1Q = False;
iEvaluateLazily = 𝕓[β]
```

Setting the value of `𝕓Equal1Q` to `True` result in `𝕓[β]` evaluating to `1`.

```
𝕓Equal1Q = True;
𝕓[β]
```

Now `iEvaluateLazily` has the stored value `𝕓[β]` (as we can see by looking at `OwnValues[iEvaluateLa⋱ zily]`) that in turn should evaluate to 1. But it does not.

```
iEvaluateLazily
```

In such cases, we can force evaluation using the function `Update[]`. In a sense, it does tell *Mathematica* to avoid stored shortcuts.

```
Update[]; iEvaluateLazily
```

Using the command `Condition`, it is possible to send a message to the user when a function is applied to a variable of the "incorrect" type. We now give an example involving a function `makeTable` requiring a positive integer for its second argument to match the pattern. If it is called with something else as a second argument, a message is sent to the user (between the `In` and `Out` lines), and the input is returned unevaluated. First, we need a new command that permits working on an expression with a changing parameter and puts the result for each value of the parameter in a list.

> `Table[`*expression*`,` *iterator*`]`
>
> produces a list of *expression* according to the iterator *iterator*.

iterator is an iterator, as discussed in detail regarding the command `Do` in Subsection 4.2.1. The following function `makeTable` issues a message when its second argument is not a positive integer.

```
makeTable::makeAll =
"The second argument must be a positive integer.";

makeTable[x_, y_] := makeTableAux[x, y] /;
                              (makeTableAux[x, y] =!= "failed")

makeTableAux[x_, y_] := If[(* is y a sensible argument? *)
                           Head[y] =!= Integer || y <= 0,
                           Message[makeTable::makeAll]; "failed",
                           Table[x, {i, y}]]

makeTable[𝒯, 3]

makeTable[𝒯, 3.0]
```

The following implementation would have given the same result, and it does not make use of an auxiliary function. However, it severely overloads `PatternTest` and thus is more difficult to understand. The message takes effect during the check of the truth value for `(If[Head[#] === Integer && # > 0, True, Message[make⋱ Table::makeAll]; False]&)`.

```
Remove[makeTable];

makeTable::makeAll =
"The second argument must be a positive integer ";

makeTable[x_, y_?(If[Head[#] === Integer && # > 0, True,
               Message[makeTable::makeAll];
               False]&)] := Table[x, {i, y}]

makeTable[𝒯, 3]
```

The message `makeTable` is printed out.

```
makeTable[𝒯, 3.0]
```

The following command for patterns is analogous to `Select`.

> `Cases[`*expression*`,` *pattern*`,` *levelSpecification*`,` *n*`]`
>
> creates a list (head `List`) of the first *n* parts of the level(s) *levelSpecification* of *expression* which match the pattern *pattern*. If *n* is not given, all parts are returned. If the level is not specified explicitly, it is taken to be 1.

Here, we are looking for all integers and occurrences of `Sin` in the levels 1 to ∞. (Note that `Sin[5 i]` has the head `Sin`.)

```
Clear[i, e, u];

Cases[e + Sin[5 i] + u 897 + Log[5678] - Exp[5/6] + 55,
      _Integer | _Sin, Infinity]
```

As with `Level` and `Position`, `Cases` also has the option `Heads`.

```
Options[Cases]
```

Note the difference between `Select` and `Cases`. `Select` picks the arguments according to the truth value, and it delivers the result with the same head as the selected expression. `Cases` chooses according to patterns, and it gives a result in the form of a list. The optional third argument in the two functions also has a completely different role. In `Select`, it defines the number of objects to be selected, whereas in `Cases`, it gives the level specification at which the first argument is to be tested.

Although arbitrary patterns can be used in function definitions with `Set` and `SetDelayed`, this does not work in pure functions. In the short form with `Slot`, no opportunity exists, and in the long form with `Function[`*arg*`, ` *f*(*arg*)`]`, the first argument must be a symbol. One of the big advantages of pure functions is that no name is needed, so what do we get if the function is not applicable?

```
Clear[f];
f = Function[x_Integer, x^2]
```

The type can be "distinguished" in such cases as follows, e.g., here using the named function `f`.

```
Clear[f];
f = Function[x, Which[Head[x] === Integer,  {x, "int"},
                      Head[x] === Rational, {x, "rat"},
                      Head[x] === Complex,  {x, "com"},
                      Head[x] === Symbol,   {x, "sym"}]];
```

Now, if the function is not applicable, we get the result `Null` (coming from `Which`).

```
{f[2], f[5I], f[o], f[3.9]}
```

In the following example, we want to find all products of squares in the list `{p^2 q^2, 2 2}`. This does not work, because the second argument is computed to be `Power[_, 2]^2` before the comparison, but this structure does not appear in `{p^2 q^2, 2 2}`.

```
Cases[{p^2 q^2, 2 2}, Power[_, 2] Power[_, 2]]
```

`Cases` does not have a `Hold`-like attribute.

```
Attributes[Cases]
```

In such situations, we have to use `HoldPattern` to get the desired result.

```
Cases[{p^2 q^2, 2^2 2^2}, HoldPattern[Power[_, 2] Power[_, 2]]]
```

(The first argument is, of course, again evaluated before any pattern matching happens. In this situation, we could have also used the pattern `a_^2` and avoided the use of `HoldPattern`.) At this point, we introduce the `Switch` command. It is related to `Which`, but works for patterns.

`Switch[`*expression*`, ` *pattern*$_1$`, ` *then*$_1$`, ` *pattern*$_2$`, ` *then*$_2$`, ` …`, ` *pattern*$_n$`, ` *then*$_n$`]`

gives the result *then*$_i$ corresponding to the first pattern matching *expression*. If none of them do, the expression remains unevaluated.

Here are two simple examples.

```
Switch[3 5/7, _Integer, "int", _Real, "rea", _Rational, "rat"]

Switch[λ + κ, _Subtract, "sub", _, 2]
```

To conclude this section on more complicated patterns, we look at a function generating such patterns from the abbreviations of the *Mathematica* commands used typically in patterns. It serves only to illustrate the multiplicity of possible patterns and allowed syntax; most of the generated patterns are not likely to be used.

```
AllSyntacticallyCorrectExpressions[
         symbolsUsed_?(VectorQ[#, StringQ]&)] :=
TableForm[(* format output nicely *)
{StringDrop[StringDrop[ToString[InputForm[#]], 5], -1],
 StringDrop[StringDrop[ToString[FullForm[#]], 5], -1]}& /@
Union[Flatten[Union[ (* test syntax *)
  If[SyntaxQ[#], ToHeldExpression[#], {}]& /@
      StringJoin /@ (* all combinations *)
Permutations[Join[#, Table[" ", {Length[#] - 1}]]]]& /@
DeleteCases[Sort[Distribute[{{}, {#}}& /@
    symbolsUsed, List, List, List, Join]], {}]]],
            TableDirections -> {Column, Row},
            TableSpacing -> {1, 3}]
```

Only the results are of interest here, not the details of the program. The argument is a list of `String`s appearing in the pattern. `AllSyntacticallyCorrectExpressions` generates patterns in which not all symbols are related, and it inserts at most one white space character between any two symbols of the argument. The output of the patterns associated with given symbols is in the form `InputForm[`*pattern*`]` and `OutputForm[`*pattern*`]`.

We now look at a few examples. (The period `.` is not only important in the commands `Optional`, `Repeated`, and `RepeatedNull`, but also as a matrix product in the form `Dot`; we come back to this in the next chapter.)

```
AllSyntacticallyCorrectExpressions[{"x", ":", "_", "."}]
```

Here is another example: We add `"&"` to the list.

```
AllSyntacticallyCorrectExpressions[{"x", ":", "_", ".", "&"}]
```

Some care should be taken when experimenting with this function; the computational time grows essentially like the factorial of (2 *numberOfSymbols*), but it is highly informative to experiment with this function.

Finally, we would like to make a remark about the possibility of writing so-called generic programs in *Mathematica*. We can give several independent function definitions on the right-hand side for a function *f*(*arguments*) corresponding to different types of arguments (which can be distinguished with `Blank[`*head*`]`, `PatternTest`, or `Condition`), which makes it possible to recursively program very complex relationships using very few function definitions (which themselves may include functions depending on the argument types). For details, see [21✱] and [22✱].

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

■ 5.2.3 Attributes of Functions and Pattern Matching

The attributes `Orderless`, `Flat`, and `OneIdentity`, which we discussed extensively in Chapter 3, have a major influence on the applicability of patterns. We begin with a discussion of the attribute `Orderless`.

Orderless

Mathematica will take into account the attribute `Orderless` in matching patterns.

We now define a function `orderlessFunction` with the attribute `Orderless`. Note that the second argument is `z`, not `z_`.

```
Remove[orderlessFunction];
SetAttributes[orderlessFunction, Orderless];
(* variable a and fixed z *)
orderlessFunction[a_, z] := {a, z};
```

In the definition of `orderlessFunction`, the "variable" variable (with `Blank`) `a` appears before the "fixed" `z`.

```
Definition[orderlessFunction]
```

Nevertheless, the definition can also be applied to arguments in the reverse order so long as exactly two arguments exist, one of which is `z`.

```
{orderlessFunction[a, z], orderlessFunction[z, a],
  (* no match for three-argument calls *)
 orderlessFunction[a, b, z]}
```

Be aware that the attributes are attached to a function that is used as a head and has or has not downvalues. The following upvalue for `x` does not result in a matching. In Chapter 4, we discussed the evaluation sequence. The attribute `Orderless` is taken into account before the upvalue of `x`.

```
Remove[x, y, z, f];
x /: f_[z, x, y] := "matched"
```

```
??x
```

```
SetAttributes[f, Orderless];
{f[x, y, z], f[x, z, y], f[y, x, z],
 f[y, z, x], f[z, x, y], f[z, y, x]}
```

Flat

If a function has the attribute `Flat`, the function is associative; *Mathematica* takes this condition into account when matching patterns. Now, we introduce a function `flatFunction` with the attribute `Flat`.

```
Remove[flatFunction, a, b, c, g];
SetAttributes[flatFunction, Flat];
flatFunction[a_, b_] := g[a, b]
```

Although it seems that, according to the definition, `flatFunction` only works for exactly two arguments, it also works for more than two arguments.

```
flatFunction[a, b, c]
```

This is because the properties of `flatFunction` are applied as often as possible.

```
flatFunction[a, b, c]
= flatFunction[flatFunction[a], flatFunction[flatFunction[b], flatFunction[c]]]
(* the definition goes into effect *)
=  g[flatFunction[a], g[flatFunction[b], flatFunction[c]]]
```

Using `On[]`, we can see some of the steps. As discussed in Chapter 4, attributes are taken into account before function definitions, so the first step is not visible here.

```
On[];
flatFunction[a, b, c]
Off[];
```

Called with two arguments, the function `flatFunction` first gets wrapped around the arguments and then the definition with `g` on the right-hand side is applied.

```
flatFunction[b, a]
```

Thus, attributes are applied in "both directions". For the purpose of matching patterns, the expression is transformed into an appropriate form in the example above by the insertion of `flatFunction`. If no explicit definition for `flat⋱Function` had been given, the appearances of `flatFunction` on the inside would have vanished in the following example.

```
Remove[flatFunction, a, b, c];
SetAttributes[flatFunction, Flat];
flatFunction[flatFunction[flatFunction[a]],
             flatFunction[flatFunction[b], flatFunction[c]]]
```

Here, we also make use of the attribute `Flat` of `flatFunction`. The `Verbatim` is needed to avoid the evaluation of `flatFunction[flatFunction[x]]` to `flatFunction[x]`.

```
Cases[{flatFunction[x]}, Verbatim[flatFunction[flatFunction[x]]]]
```

But in some cases, recursion problems with `Flat` might occur, so some caution is in order. For instance, in the following example, `x` is replaced by `f[x]` during pattern matching, so the function definition involves an infinite loop.

```
Remove[f]
SetAttributes[f, Flat]
f[x_f] := x;
f[x]
```

A pattern of the form `x_` in a function definition with the attribute `Flat` matches more than one argument.

```
Remove[f, x, y]
SetAttributes[f, {Flat}];
f[x_] := Matched[Hold[x]]
{f[], f[x], f[x, x], f[x, y], f[x, y, x]}
```

We now turn to the attribute `OneIdentity`.

OneIdentity

We examine the previous example to illustrate the two attributes `Flat` and `OneIdentity`. `flatOneIdentity⋱Function[`*x*`]` is identical to *x* with respect to matching of patterns (again, we give no explicit definition for `flatOne IdentityFunction`).

```
Remove[flatOneIdentityFunction, a, b, c, g];
SetAttributes[flatOneIdentityFunction, {Flat, OneIdentity}];
flatOneIdentityFunction[a_, b_] := g[a, b]

flatOneIdentityFunction[a, b, c]
```

In comparison with the `flatFunction` example, the inner `flatOneIdentityFunction` appearances are missing; however, `g` appears on the inside. `OneIdentity` is very important for the application of default values (the command `Default` was still missing in our discussion of `Optional`). First, we show how these default values can be defined.

`Default[`*function*`] = ` *value*

assigns the default value *value* to the function *function*. It is used for definitions of the form *function*`[..., ` *variable*`_.]`. `Default[`*function*`]` should be defined before the definition of *function*, and after the assignment of attributes.

Just two built-in functions have a predefined default value.

```
Select[Names["*"], (Head[Default[#]] =!= Default)&]
```

These two default values cause `Plus` and `Times` with one argument to evaluate to the argument.

```
{Default[Plus], Default[Times]}
```

```
{Plus[Z], Times[Z]}
```

First, we give a simple example for the use of `Default`, without specifying any attributes.

```
Remove[f];
Default[f] = a[l][w][a][y][s];
f[x_, y_.] := {x, y};
{f[x, y], f[x]}
```

Here, the first argument is optional.

```
Remove[f];
Default[f] = a[l][w][a][y][s];
f[x_., y_] := {x, y};
{f[x, y], f[x]}
```

Here is a somewhat more complicated one. We define *def* as a function with the attribute `OneIdentity`, and the default value 123456789.

```
Remove[def];
SetAttributes[def, OneIdentity];
Default[def] = 123456789;
```

We now define a function `functionWithDefaultValue` containing `def`. We associate the definition with `funcWithDef`.

```
Remove[functionWithDefaulValue];
functionWithDefaultValue[def[x_, y_.]] := ℍ[x, y]
```

Next, we give some examples to illustrate the behavior of the function `functionWithDefaultValue`. With two arguments, we do not get the default value, and the attribute `OneIdentity` plays no role.

```
functionWithDefaultValue[def[4, 4]]
```

With one argument, the default value is used for the second argument.

```
functionWithDefaultValue[def[a]]
```

Next, we call `functionWithDefaultValue` with `a` as a direct argument, without *def*`[a]`. For the purposes of the pattern recognition, `a` is equivalent to *def*`[a]`, which in view of the default value of *def* is equivalent to *def*`[a, 123456789]` (because of the `OneIdentity` attribute). Therefore, the result is `ℍ[a, 123456789]`.

```
functionWithDefaultValue[a]
```

For comparison, let us use the same definitions without the `OneIdentity` attribute of `def`.

```
Remove[def, functionWithDefaultValue];
Default[def] = 123456789;
functionWithDefaultValue[def[x_, y_.]] := ℍ[x, y];
functionWithDefaultValue[a]
```

Nothing happened. To summarize: `Orderless` and `Flat` can cause expressions to evaluate differently. `OneIdentity` has only effects when used in pattern-matching situations. In two instances, the `OneIdentity` attribute matters: a) in connection with the `Flat` attribute and b) in connection with default values (`Default` and `Optional`).

`OneIdentity` and `Flat`

Because of the importance of the attribute combination `Flat` and `OneIdentity`, we will discuss this duo separately. The effect of the attribute `OneIdentity` is often expressed as "a, f`[`a`]`, f`[`f`[`a`]]` are equivalent for pattern matching". But this sentence is not to be interpreted literally!

```
Remove[f]
SetAttributes[f, OneIdentity];
{MatchQ[x, f[x]], MatchQ[f[x], f[f[x]]]}
```

Here is case a) demonstrated.

```
Remove[f]
SetAttributes[f, Flat];
f[__, _String, __] := "yes"
f["a", "b", "c", "d"]
```

```
Remove[f]
SetAttributes[f, {Flat, OneIdentity}];
f[__, _String, __] := "yes"
f["a", "b", "c", "d"]
```

Let us use a side effect to see what happens. If a function *f* has the `Flat` and `OneIdentity` attribute, single elements will not be wrapped in *f* before trying to match.

```
Remove[f]
SetAttributes[f, Flat];
f[a__, b_, c__] /;
         (Print[{{a}, {b}, {c}}]; Head[b] === String) := yes
f["a", "b", "c", "d"]
```

```
Remove[f]
SetAttributes[f, {Flat, OneIdentity}];
f[a__, b_, c__] /;
         (Print[{{a}, {b}, {c}}]; Head[b] === String) := yes
f["a", "b", "c", "d"]
```

And here is an example of case b). In this case, the `Flat` attribute has no effect.

```
Remove[f, 𝕡];
f[𝕡[_:0]] := "yes";
f[1]
```

```
Remove[f, 𝕡];
SetAttributes[𝕡, {OneIdentity}];
f[𝕡[_:0]] := "yes";
f[1]
```

```
Remove[f, ℙ];
SetAttributes[ℙ, {Flat, OneIdentity}];
f[ℙ[_:0]] := "yes";
f[1]
```

We now look at these three combinations using an example of a function with all three attributes: `Orderless`, `Flat`, and `OneIdentity`.

`Flat`, `OneIdentity`, and `Orderless`

If a function has the attributes `Orderless`, `Flat`, and `OneIdentity`, *Mathematica* takes into account all three of these attributes when matching patterns. Consider the following function `hAV` (short for has various attributes) with the attributes `Orderless`, `Flat`, and `OneIdentity`. To better compare the effect of the individual attributes when all three are assigned, in the following example all possible variants of the assignment of attributes are presented (i.e., `Orderless`, `Flat`, and `OneIdentity` by themselves, in pairs, and all three together). We recommend that the reader goes carefully through all inputs and outputs and try, in all cases, to understand what happened.

```
Remove[hAV, a, b, c, d];
SetAttributes[hAV, {Orderless}];
hAV[x_, x_] := Z[x];
{hAV[a], hAV[a, a], hAV[a, b, c, d, a]}

{hAV[a], Z[a], hAV[a, a, b, c, d]}

Remove[hAV, a, b, c, d];
SetAttributes[hAV, {OneIdentity}];
hAV[x_, x_] := Z[x];
{hAV[a], hAV[a, a], hAV[a, b, c, d, a]}

{hAV[a], Z[a], hAV[a, b, c, d, a]}

Remove[hAV, a, b, c, d];
SetAttributes[hAV, {Flat}];
hAV[x_, x_] := Z[x];
{hAV[a], hAV[a, a], hAV[a, b, c, d, a]}

{hAV[a], Z[hAV[a]], hAV[a, b, c, d, a]}

Remove[hAV, a, b, c, d];
SetAttributes[hAV, {Orderless, Flat}];
hAV[x_, x_] := Z[x];
{hAV[a], hAV[a, a], hAV[a, b, c, d, a]}

{hAV[a], Z[hAV[a]], hAV[b, c, d, Z[hAV[a]]]}

Remove[hAV, a, b, c, d];
SetAttributes[hAV, {OneIdentity, Flat}];
hAV[x_, x_] := Z[x];
{hAV[a], hAV[a, a], hAV[a, b, c, d, a]}

{hAV[a], Z[a], hAV[a, b, c, d, a]}

Remove[hAV, a, b, c, d];
SetAttributes[hAV, {Orderless, OneIdentity}];
hAV[x_, x_] := Z[x];
{hAV[a], hAV[a, a], hAV[a, b, c, d, a]}

{hAV[a], Z[a], hAV[a, a, b, c, d]}
```

Here is the most interesting case with all three attributes present.

```
Remove[hAV, a, b, c, d];
SetAttributes[hAV, {Orderless, OneIdentity, Flat}];
hAV[x_, x_] := Z[x];
{hAV[a], hAV[a, a], hAV[a, b, c, d, a]}
```

Here is what happened with `hAV[a, b, c, d, a]` in the last input. Because of the `Orderless` attribute, it is converted to `hAV[a, a, b, c, d]`. Then, the attribute `Flat` gives `hAV[hAV[hAV[a],hAV[a]], b, c, d]`.
With the `OneIdentity` and `Flat` attribute, this becomes `hAV[hAV[a, a], b, c, d]`. Then, by the definition of `hAV`, we get `hAV[Z[a], b, c, d]`. Another application of the `Orderless` attribute leads to `hAV[b, c, d, Z[a]]`.
`On` does not give us much information about the use of attributes.

```
On[]; hAV[a, b, c, d, a]; Off[]
```

The same remark goes for `Trace`.

```
Trace[hAV[a, b, c, d, a]]
```

But we can associate a rule with `a` that every function containing `a` is printed together with its arguments.

```
Remove[hAV, a, b, c, d];
a /: 𝕗:(f_[___, a, ___]) := Null /; (Print[HoldForm[𝕗]]; False);
SetAttributes[hAV, {Orderless, OneIdentity, Flat}];
hAV[x_, x_] := Z[x];
{hAV[a], hAV[a, a], hAV[a, b, c, d, a]}

{hAV[a], Z[a], hAV[b, c, d, Z[a]]}
```

For later use of the variable `a`, we remove the rule attached to `a`.

```
Remove[a]
```

This example shows that if a function has several attributes, the matching of patterns can be very complicated.

We now give an "automated" example to illustrate the way in which the attributes of functions (`Orderless`, `Flat`, and `OneIdentity`) work together with `Blank`, `BlankSequence`, or `BlankNullSequence` in matching patterns. To save some writing, we define a function `patternsAndAttributes`. Its first argument contains the attributes, and its second contains the blanks for a function to be generated in the form `g[x_, Pattern[y,` *blanks*`]]`. We call this function `g` with three arguments `g[x, y, z]`, and look at the interpretation of `x` and `y` selected by *Mathematica*. To get all possible interpretations of `x` and `y`, we define `g` under a condition that is never satisfied (`False`), and in addition, write out the three arguments. We apply `Block` to override `$RecursionLimit` locally in case something goes wrong. To avoid a reevaluation of the matched variables in the right-hand side, we enclose them in a `Hold`.

```
Remove[PatternsAndAttributes, x, y];

PatternsAndAttributes[attris_, blanks_] :=
Block[{g, nothing, $RecursionLimit = 20},
  SetAttributes[g, attris];
  (* the function definition is generated here *)
  SetDelayed[Evaluate[g[x_, Pattern[y, blanks]]], Condition[nothing,
                Print["x ⟶ ", Hold[x], "  y ⟶ ", Hold[y]]; False]];
  g[x, y, z]; ]
```

Here is one example.

```
PatternsAndAttributes[{Orderless}, __]
```

To avoid a large amount of output, we will not look at all of the tried pattern matchings, but only at the number of tried patterns. The next version of the function `patternsAndAttributes` returns the numbers of matchings.

```
Remove[PatternsAndAttributes, x, y];

PatternsAndAttributes[attris_, blanks_] :=
Block[{g, nothing, bag = {}, $RecursionLimit = 20},
      SetAttributes[g, attris];
     (* the function definition is generated here *)
       SetDelayed[Evaluate[g[x_, Pattern[y, blanks]]],
            Condition[Null,
             (* collect matchings *)
             AppendTo[bag, {"x⟶", HoldForm[x],
                            "y⟶", HoldForm[y]}]; False]];
      g[x, y, z]; Length[bag]]
```

For a given list of attributes *attris*, we will test all three patterns.

```
numberOfTrials[attris_] :=
{PatternsAndAttributes[attris, _],
 PatternsAndAttributes[attris, __],
 PatternsAndAttributes[attris, ___]}

numberOfTrials[{Orderless}]

numberOfTrials[{Flat}]

numberOfTrials[{OneIdentity}]

numberOfTrials[{Orderless, Flat}]

numberOfTrials[{Orderless, OneIdentity}]

numberOfTrials[{Orderless, Flat, OneIdentity}]
```

In this subsection, we discussed the interaction of pattern matching with the attributes `Orderless`, `Flat`, and `OneIdentity`. These three attributes are most important with respect to pattern matching. But other attributes (such as `Hold`-like ones) are sometimes of relevance too. The following example shows a function with the two attributes `Orderless` and `HoldAll` at work.

```
Remove[𝕗];
SetAttributes[𝕗, {Orderless, HoldAll}];
𝕗[x_ + x_, x_] := x
𝕗[2 + 2, 2 + 2 + 2 + 2]
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

5.3 Replacement Rules

■ 5.3.1 Replacement Rules for Patterns

Mathematica includes several ways to replace certain parts of expressions by others. This feature is very important for "manual manipulation and simplification" of expressions. The simplest is `Rule`.

`Rule[`*beforehand*`, `*afterward*`]`

or

beforehand `->` *afterward*

represents the replacement rule, which replaces the expression *beforehand* with the expression *afterward* when it is applied to an expression. *beforehand* can contain patterns.

At the point when this command is executed, both the left- and right-hand sides of the input are evaluated to the furthest extent possible using this rule.

```
{1 2 3 -> t t z, 2 x_ 3 -> 3^(3 4z_)}
```

Analogous to `Set` and `SetDelayed`, it may be necessary to compute only at the time when making the replacement. (Recall the example of `SetDelayed` involving `Expand` in Subsection 3.1.1.) The equivalent to `SetDelayed` for rules is `RuleDelayed`.

`RuleDelayed[`*beforehand*`,` *afterward*`]`

or

beforehand `:>` *afterward*

represents the replacement rule, which when applied to an expression, replaces the expression *beforehand* with the expression *afterward* (*afterward* is then evaluated at the time of the application of the rule). *beforehand* can contain patterns.

The fact that *afterward* is computed at a later point can be seen by looking at the attributes of `Rule` and `RuleDelayed`.

```
Attributes[Rule]
```

```
Attributes[RuleDelayed]
```

Using a similar example as above, we see that the right-hand sides remain unevaluated.

```
{1 2 3 :> t t z, 2 x_ 3 :> 3^(3 4z_)}
```

The application of the replacement rules is accomplished with one of the following three commands: `Replace`, `ReplaceAll`, and `ReplaceRepeated` (or with the command `StringReplace` discussed in Section 4.4).

`Replace[`*expression*`,` *rules*`]`

carries out the replacement rules *rules* on the expression, in which *rules* is applied only to the entire *expression*. Here, *rules* is of type `Rule` or `RuleDelayed`, or it is a (possibly nested) list of such expressions.

`ReplaceAll[`*expression*`,` *rules*`]`

or

expression `/.` *rules*

carries out the replacement rules *rules* on *expression*, in which each rule in rules is applied just once to each subexpression of *expression*. Here, *rules* is a rule of type `Rule` or `RuleDelayed`, or it is a (possibly nested) list of such expressions.

`ReplaceRepeated[`*expression*`,` *rules*`]`

or

expression `//.` *rules*

carries out the replacement rules *rules* on expression, in which *rules* is applied to all subexpressions until *expression* no longer changes. Here, *rules* is of type `Rule` or `RuleDelayed`, or it is a (possibly nested) list of such expressions.

The following example illustrates the differences between the three commands. We start with an expression called `expression`.

```
Remove[expression, xu, yu, xo, yo, f, exp, f𝔸, add];

expression = (xu^xu + yu^yu)^(xo^xo + yo^yo) + 1
```

Because `Replace` only manipulates the entire expression, nothing happens with the replacement rule `b_^exp_ -> f[b, exp]` (*expression* has the structure `1 + b_^c_`, not `b_^c_`.).

```
Replace[expression, b_^exp_ -> f[b, exp]]
```

`Replace` works with the rule `b_^exp_ + add_`.

```
Replace[expression, b_^exp_ + add_ -> f𝔸[b, exp, add]]
```

`ReplaceAll` manipulates every subexpression just once. Note that the rule is not applied to subsubparts of a subexpression that successfully would have matched the pattern.

```
ReplaceAll[expression, b_^exp_ -> f[b, exp]]
```

`ReplaceRepeated` is applied as often as possible.

```
ReplaceRepeated[expression, b_^exp_ -> f[b, exp]]
```

By adding a print statement on the right-hand side of the rule and making sure that the rule never applies, we see in which order the various parts of `expression` are tried in the pattern-matching process and replacing process.

```
rule = part_ :> (Null /; (Print["Trying : ", InputForm[part]]; False))

expression /. rule
```

Here is a more complicated example of the application of `ReplaceRepeated`. All `M`s in the summand of the following expression should be collected in `MContainers`. We count how often a rule is used by `counter`.

```
(* initialize counter  *)
counter = 0;

(2 a M[1] M[2] M[] + 3 b  M[1, 3] M[2] M["s"] +
 Log[f[M[1] M[1.2] M[3.4] M[M]]]) //. (* the rules *)
           {m1_M m2_M :> (counter = counter + 1; MContainer[m1, m2]),
            MContainer[m1__] m2_M :> (counter = counter + 1;
                                      MContainer[m1, m2]),
            MContainer[m1__] MContainer[m2__] :>
                 (counter = counter + 1; MContainer[m1, m2])}

Print[counter]; Remove[counter]
```

Note the behavior of the pattern `BlankNullSequence`; it gives `Sequence[]` in the following example. The "empty" argument(s) of `{}` is extracted.

```
{} /. {a___} -> a
```

The replacement rules inside of `Replace`, `ReplaceAll`, and `ReplaceRepeated` must be given in the form of a list when several components exist.

```
f[a, b, c, d] /. {a -> 1, b -> 2, c -> 3, d -> 4}
```

If the list is empty, nothing happens.

```
f[a, b, c, d] /. {}
```

Note that replacements using `Replace`, `ReplaceRepeated`, or `ReplaceAll` also take place inside functions carrying attributes like `Hold`. So the result of the following input is not `{ζ[0],ζ[1],ζ[2],ζ[3],ζ[4],ζ[5]}`.

```
SetAttributes[ζ, HoldAll]
Table[ζ[i], {i, 0, 5}]
```

By using a replacement rule to substitute the values of the iterator variable, we can go inside `tz`.

```
Table[ζ[k] /. k -> i, {i, 0, 5}]
```

Here are some similar examples.

```
({#1, #2}&) /. #2 -> #1
```

```
(x :> 5) /. (x :> 6)
```

```
Hold[2 + 2] /. {2 -> 3}
```

```
SetAttributes[f, HoldAllComplete];
f[2 + 2] /. {2 -> 3}
```

Note that the `1 + 1` in the following example gets replaced and that the `2 + 2` was never evaluated.

```
Hold[1 + 1] /. {HoldPattern[1 + 1] :> 2 + 2}
```

In the last input example, the curly braces are needed as a container for the rules because the decimal point binds more strongly. If we use appropriate formatting with white space around low-binding operators, this problem does not exist.

```
Hold[2 + 2] /.2 -> 3
```

`FullForm` makes clear what happened.

```
DownValues[In][[-2]] // FullForm
```

Alternatively, we could follow our formatting conventions.

```
Hold[2 + 2] /. 2 -> 3
```

```
Attributes[HoldPattern]
```

```
HoldPattern[x_ + y_] /. y -> z
```

Here, the use of `HoldPattern` to get the desired replacement is unavoidable.

```
HoldForm[1 + (2 3) + 4 Sin[3 + 6 Nis[5 6]]] /. {5 6 -> 6 5}
```

```
HoldForm[1 + (2 3) + 4 Sin[3 + 6 Nis[5 6]]] /.
                    {HoldPattern[5 6] -> HoldForm[6 5]}
```

This example is similar. Because of the `HoldForm` enclosing `Nis`, we do not need an additional `HoldForm` on the right-hand side of a delayed rule.

```
HoldForm[1 + (2 3) + 4 Sin[3 + 6 Nis[5 6]]] /.
                              {HoldPattern[5 6] :> 6 5}
```

But with `Unevaluated` instead of `HoldForm`, we get a somewhat different result. `Unevaluated` has the `Hold`⋱`All` attribute. This attribute avoids that the arguments are getting evaluated before they are passed to the enclosing function. And `ReplaceAll` will respect the `Unevaluated` fully and not evaluate its argument; else the pattern `5 6` would have been not present anymore. The next input shows that `5 6` was really replaced and after the replacement, the products evaluated.

```
Unevaluated[1 + (2 3) + 4 Sin[3 + 6 Nis[5 6]]] /.
                              {HoldPattern[5 6] :> 6 7}
```

If we have an expression of the form *expression* `/.` *rule* (the `FullForm` would be `ReplaceAll[`*expression*`,` *rule*`]`), by the order of calculation discussed in Chapter 4, the first *expression* is computed, and then the replacement rule is carried out. Thus, the result of `(2 - 1 - 1) /. {-1 -> 11}` is 0, and not 24.

```
(2 - 1 - 1) /. {-1 -> 11}
```

Avoiding standard evaluation, we can produce the result 24.

```
Unevaluated[2 - 1 - 1] /. {-1 -> 11}
```

There are two `-1` present in the unevaluated form of `2 - 1 - 1`.

```
Unevaluated[2 - 1 - 1] // FullForm
```

For first time users of replacement rules, we often do not get the desired replacement. Here is an example. We start with a simple fraction.

```
Clear[a, b, c];

a/b^2
```

We want to substitute `c^2` for `b^2`.

```
% /. {b^2 -> c^2}
```

This is another example.

```
a + (2 + I) b
```

We want to substitute `-I` for `I`.

```
% /. {I -> -I}
```

Neither substitution works, because the subexpressions that are to be replaced do not appear in the form given in the replacement rule. We can see the structure of an expression best with `FullForm`.

```
FullForm[a/b^2]

FullForm[a + (2 + I) b]
```

In these two cases, this rule would have been suitable.

```
a/b^2 /. {b^-2 -> c^-2}

a + (2 + I) b /. {2 + I -> 2 - I}
```

Similarly, `x + 1 + (x^2 - 1)/(x - 1) //. {1 + x -> y}` does not give `2y`, because `(x^2 - 1)/(x - 1)` is not simplified to `x + 1` in any step of the calculation.

Here is another frequently occurring situation of a nonmatching pattern. The following integral returns an `If` (the first element of `If` describes the range of the parameters that guarantee convergence).

```
Integrate[x^-α + x^α, {x, 0, Infinity}]
```

But the following replacement does not work.

```
% /. (0 < Re[α] < 1) -> True
```

The reason is that inside the `If` statement, we have an expression with head `Inequality`, but `0 < Re[α] < 1` is parsed as an expression with head `Less`.

```
{%%[[1]], 0 < Re[α] < 1} // FullForm
```

Using `Inequality` in the replacement the `If` evaluates to its first argument.

```
%%% /. (Inequality[0, Less, Re[α], Less, 1]) -> True
```

Because `OutputForm`, `StandardForm`, and especially `TraditionalForm` often differ considerably from the `FullForm`, it can be very useful to examine the `FullForm` of the expression when applying replacement rules.

Similarly, a replacement often fails because the replacement rule is not applied to the desired part of the expression. For example, suppose we want to replace all `f[`*something*`]` expressions, except for the outermost one, by `h[`*something*`]` in `f[f[x, f[x]], f[x, f[x]]]`. The following example does not accomplish this goal.

```
Clear[f, h, x];
f[f[x, f[x]], f[x, f[x]]] /. f[x__] -> h[x]
```

`ReplaceRepeated` replaces all `f` appearing in the expression, including the outermost one.

```
f[f[x, f[x]], f[x, f[x]]] //. f[x__] -> h[x]
```

But the next input does work (see the next section). The idea is to map the replacement rule inside the expression.

```
(# //. f[x__] -> h[x])& /@ f[f[x, f[x]], f[x, f[x]]]
```

Sometimes, we want all possible matchings for a certain pattern. The function `ReplaceList` gives such a list.

`ReplaceList[`*expression*`, `*replacementRules*`]`
returns a list of all possible results of applying the replacement rules *replacementRules* to *expression*.

No matches are found in the following example. `{1, 2, 3, 4, 5, 6}` has length six and the pattern `{a_, b_, c_}` specifies a list of length three.

```
ReplaceList[{1, 2, 3, 4, 5, 6},
            {a_, b_, c_} :> {{a}, {b}, {c}}]
```

Now, we have 10 matchings.

```
ReplaceList[{1, 2, 3, 4, 5, 6},
            {a__, b__, c__} :> {{a}, {b}, {c}}]

Length[%]
```

Using `BlankNullSequences` instead of `BlankSequences` results in 28 matchings.

```
ReplaceList[{1, 2, 3, 4, 5, 6},
            {a___, b___, c___} :> {{a}, {b}, {c}}]

Length[%]
```

In the example above, for a growing number of list elements and a fixed pattern, the number of possible matchings grows relatively slowly.

```
Table[Length[ReplaceList[Range[i],
              {a___, b___, c___} :> {{a}, {b}, {c}}]],
      {i, 20}]
```

Fixing the list and changing the replacement rules is done in the following. Because `BlankSequence` requires at least one element to match, we get the following first growing and then decreasing number of matches. (We will discuss the shortcuts `@@` and `/@` in the next chapter.)

```
Table[Length[ReplaceList[{1, 2, 3, 4, 5, 6, 7, 8, 9, 10},
(* make patterns *)
(RuleDelayed @@ {Pattern[#, BlankSequence[]]& /@ #, #})&[
            Take[{a1, a2, a3, a4, a5, a6, a7, a8, a9, a10}, i]]]],
      {i, 10}]
```

In the following example, the attributes of `Times` result in six possible replacements.

```
ReplaceList[𝕩 𝕪 𝕫, α_ β_ :> {α, β}]
```

If we use `BlankNullSequence` instead, we get an exponential growth in the number of possible matchings. (It is wise to have this exponential growth in mind when writing complicated pattern-matching based programs.)

```
Table[Length[ReplaceList[{1, 2, 3, 4, 5, 6, 7, 8, 9, 10},
(* make patterns *)
(RuleDelayed @@ {Pattern[#, BlankNullSequence[]]& /@ #, #})&[
             Take[{a1, a2, a3, a4, a5, a6, a7, a8, a9, a10}, i]]]],
      {i, 10}]
```

The way in which *Mathematica* applies replacement rules can be seen in detail in the following example, which includes three `Print` statements. The first `Print` takes effect when the parts to be replaced are encountered; the second `Print` takes effect when `PatternTest` is applied to the rule; and the third `Print` takes effect when the condition (implemented with `Condition`) is checked for the applicability of the rule. The condition is never satisfied, and so all possible replacements are investigated. First, in the pattern-matching process, the whole expression is checked, then the enclosing list, and so on.

```
(* body *)
{Unevaluated[Print["Argument 1 evaluated"]; 1],
 Unevaluated[Print["Argument 2 evaluated"]; 2],
 Unevaluated[Print["Argument 3 evaluated"]; 3]} /.
 (* replacement rules *)
  {i_?((* lhs pattern test *)
      NumberQ[Print["PatternTest of: ", #]; #]&) :> (i /;
                                (* condition on pattern *)
                                (Print["Test of ", i]; False))}
```

Similarly to the functions `Set` and `SetDelayed`, the two functions `Rule` and `RuleDelayed` respect the local binding of variables in named patterns. Here this is demonstrated.

```
Block[{x = 𝕏}, x[x_, _x, x, x_x] -> x]

Module[{x = 𝕏}, x[x_, _x, x, x_x] -> x]

With[{x = 𝕏}, x[x_, _x, x, x_x] -> x]

Function[x, x[x_, _x, x, x_x] -> x][𝕏]
```

Note that nested rules scope pattern variables through the outermost rule. So all `y_` in the following input are bound by the `Rule` with `C` on the left-hand side.

```
With[{a = 𝕩}, Hold[C[y_, y_ -> y, y_ -> (y_ -> y),
                     (y_ -> y) -> y]] -> a]
```

The outer `With` was needed to force a renaming.

```
Hold[C[y_, y_ -> y, y_ -> (y_ -> y), (y_ -> y) -> y]] -> a
```

Now, we come to the relationship between rule application and attributes.

> Attributes are taken into account when applying replacement rules to functions.

This use of attributes in replacement rules is analogous to the interaction of function definitions, attributes, and patterns discussed earlier. It should suffice to give a few examples. We do not discuss these examples in great detail. The way the results arise should become obvious after a short study.

Attribute Orderless

```
Remove[f, a, b, c, d];
```

```
SetAttributes[f, Orderless];
```

```
f[a, b] /. {f[b, a] -> d}
```

Attribute Flat

```
Remove[f, a, b, c, d];
SetAttributes[f, Flat];
```

Here, `f[a]` has to be interpreted as `f[f[a], f[]]` to match the pattern.

```
{f[a] /. {f[] -> {d}},
 f[a] /. {f[d_] -> {d}},
 f[a] /. {f[d__] -> {d}}}
```

```
{f[a, b] /. {f[] -> {d}},
 f[a, b] /. {f[d_] -> {d}},
 f[a, b] /. {f[d__] -> {d}}}
```

```
{f[a, b, c] /. {f[] -> {d}},
 f[a, b, c] /. {f[d_] -> {d}},
 f[a, b, c] /. {f[d__] -> {d}}}
```

```
Replace[f[f[a], a, a, f[a]], f[a___] -> b]
```

Attribute OneIdentity

```
Remove[f, a, b, c];
SetAttributes[f, {OneIdentity}];
```

```
{f[f[a]] /. {f[a] -> {d}},
 f[a] /.  {f[f[a]] -> {d}},
 a /. {f[a] -> {d}},
 f[a_:1] /. {1 -> {d}},
 1 /. {f[a_:2] -> {d}}}
```

Attributes Flat and OneIdentity

```
Remove[f, a, b, c];
SetAttributes[f, {Flat, OneIdentity}];
```

```
{f[f[a]] /. {f[a] -> {d}},
 f[a] /. {f[f[a]] -> {d}},
 a /. {f[a] -> {d}}}
```

```
{f[a] /. {f[] -> {d}},
 f[a] /. {f[d_] -> {d}},
 f[a] /. {f[d__] -> {d}},
 f[f[f[a]], f[a], a] /. {f[a] -> {d}}}
```

```
{f[a, b] /. {f[] -> {d}},
 f[a, b] /. {f[d_] -> {d}},
 f[a, b] /. {f[d__] -> {d}}}
```

```
{f[a, b, c] /. {f[] -> {d}},
 f[a, b, c] /. {f[d_] -> {d}},
 f[a, b, c] /. {f[d__] -> {d}}}
```

However, take note of the following rule in connection with matching patterns.

> Attributes of the function *f* affect the matching of patterns only if *f* appears as a head in the rule.

The following example works.

```
Remove[f, a, b, g];
SetAttributes[f, Orderless];
g[f[a, b]] /. {_[f[b, a]] -> "O.K."}
```

But the next input does not, even though _ is a “special case” of `f`.

```
Remove[f];
SetAttributes[f, Orderless];
g[f[a, b]] /. {_[_[b, a]] -> " works"}
```

Here is an analogous example for the attribute `Flat`.

```
Remove[f];
SetAttributes[f, Flat];
{f[a] /. {f[f[a]] -> " O.K. "},
 f[a] /. {_[f[a]] -> " works"},
 f[a] /. {f[_[a]] -> " works"},
 f[a] /. {_[_[a]] -> " works"}}
```

> It is easy to get into infinite loops using `ReplaceRepeated`. It involves iterations (not recursions), and it is applied 4096 times. (This amount is considerably more than with recurrences, and it can take a long time until it is reached.)

Currently, here are the values for `$RecursionLimit` and `$IterationLimit`.

```
{$RecursionLimit, $IterationLimit}
```

Here is an example. The following construction leads to an infinite loop.

```
Clear[i];
(1 + i) //. i -> i + 1
```

We can avoid the infinite loop by constraining the applicability of the rule: Starting with `1 + i`, we replace all sums (head `Plus`) by themselves plus 1 as long as the numerical part is smaller than 5. (Note the completely different meaning of `i` in `(i + 1)` and in `i_Plus?(Select[#, NumberQ] < 5&) :> i + 1`.)

```
(1 + i) //. i_Plus?(Select[#, NumberQ] < 5&) :> i + 1
```

For a better understanding of the last input, it is useful to look at the `FullForm`.

```
FullForm[Hold[(1 + i) //. i_Plus?((Select[#, NumberQ] < 5)&) :> (i + 1)]]
```

> If the replacement rules are in a nested `List`, the result of applying the rules will have an equivalent `List` structure.

Here is the simplest form of a replacement rule.

```
Clear[f, a, g, b];
f /. f -> a
```

If we had several replacement rules, they would have to be collected in a list. In this case, no additional brackets are around the resulting `a`.

```
f /. {f -> a}
```

Now, `List` is applied to this result once.

```
f /. {{f -> a}}
```

The same result happens in this case.

```
f /. {{f -> a, g -> b}}
```

The next substitution even leads to two pairs of braces.

```
f /. {{{f -> a}}}
```

However, if the individual replacement rules have multiple brackets, the overall structure of the replacement list is applied to the expression in which the replacement is to take place. The replacement increases the length of the result by a corresponding amount even when no replacements in the expression are possible using the given rules.

```
f /. {f -> a, g -> b}
```

```
f /. {{f -> a}, {g -> b}}
```

```
f /. {{{f -> a}, {g -> b}}}
```

```
f /. {{{f -> a}}, {{g -> b}}}
```

The next input contains rules inside lists of different depths. Be aware that the nonmatching rules caused additional lists around the `f`.

```
f /. {{f -> a}, {{g -> b}}, {{{h -> c}}}}
```

Now that we are acquainted with the commands `RuleDelayed` and `HoldPattern`, we come back to `DownValues`, which was mentioned in Chapter 3. In contrast to `Definition`, it produces the internal form used by *Mathematica* in the definition of functions.

```
Clear[f, x];
f[x_Integer] = {x^2};
f[x_Rational] := {Numerator[x], Denominator[x]}
```

```
??f
```

```
Definition[f]
```

```
DownValues[f]
```

Now, we can understand the way in which function definitions work. Internally, no difference exists in function definitions using `Set` or `SetDelayed`. Both sides of the internal function definition are stored completely unevaluated. The `HoldAll` attribute of `RuleDelayed` prevents the calculation on the right-hand side, and `HoldPattern` prevents it on the left-hand side. `HoldPattern` is necessary to suppress the computation of the arguments of `f` on the left-hand side. We can model

f[*x*_] = *functionOfx*

f[*specialValue*]

as follows: `ReleaseHold[Hold[`*functionOfx*`] /. x ->` *specialValue*`]`.

Now, for example, we can in detail understand why the following construction does not work.

```
Remove[f, g, x];
```

```
f[x_] := Module[{g}, g[x_] = x^2; g[x]];
```

```
DownValues[f]
```

```
f[1]
```

The x on the left-hand side is associated with the x on the right-hand side, and it is not applied locally in the function definitions of g.

```
Trace[f[1]]
```

But if the argument of f is a symbol, all works fine.

```
Clear[y];
f[y]
```

The same result would have happened with `g[x_] := Block[h, h[x_] = x^2; h[x]]`.

Finally, we complete the discussion of the operation and application of options. The selection of special options using `->` is just the application of a Rule-object to an expression. We first define a function testFunctionWithOptions with the two options who and time.

```
Remove[testFunctionWithOptions];
Options[testFunctionWithOptions] = {who -> myself, time -> now}
```

To later use other special settings of the two options, who and time, the following construction is necessary.

```
testFunctionWithOptions[x_, opts___] :=
   {x, who, time} /. {opts} /. Options[testFunctionWithOptions]
```

Note the grouping to give multiple replacement rules.

```
FullForm[Unevaluated[a /. b /. c]]
```

A number of interesting details are in this construction.

■ The last argument of testFunctionWithOptions has the form Pattern[*opts*, BlankNullSequence[]]. BlankNullSequence covers the case in which no special values are given for the settings and the case in which several are prescribed.

■ To put into effect the given settings in *Settings* inside of testFunctionWithOptions[*argument*, *optionsAndSettings*], we need the construction *expression* /. {*optionsAndSettings*} /. Options[*command*]. Note that first *optionsAndSettings* goes into effect in *expression* /. {*optionsAndSettings*}, and then if options still exist in the transformed *expression*, the global default takes effect via the afterward-applied set of options from Options[*command*]. Moreover, *optionsAndSettings* must be included in a list; a direct extraction would have the head Sequence, but multiple replacement rules must be input into lists. If no *optionsAndSettings* is given, {Sequence[]} (= {}) and Options[*command*] go into effect on the unchanged expression.

To make testFunctionWithOptions a bit safer with respect to possible given arguments, we could restrict the head of opts via opts___Rule | x___RuleDelayed.

We now show that the construction above works correctly.

```
testFunctionWithOptions["discussion"]
```

```
testFunctionWithOptions["discussion", time -> "5 pm"]
```

```
testFunctionWithOptions["discussion", who -> "Amy"]

testFunctionWithOptions["discussion", who -> "Amy", time -> "17.00"]
```

In the next input, the option `who` is set twice. The first option `"Amy"` takes precedence.

```
testFunctionWithOptions["discussion", who -> "Amy", who -> "Roger"]
```

We discuss `ReplacePart` as the last subject of the discussion of replacing elements. Often, it is useful to replace single elements in a larger expression (e.g., elements in a matrix). This procedure is done with `ReplacePart`.

> `ReplacePart[`*expression*`, `*newExpression*`, {`*position*`}]`
> replaces the element *expression*`[[`*position*`]]` by *newExpression*.

Here is an expression.

```
expr = 3 + Sin[5]^3 + u^3 + Log[6]
```

The exponent 3 in `u^3` is to be replaced by `new3Exp`.

```
ReplacePart[expr, expr, {2, 2}]
```

Of course, in this case, we could have also used these alternatives.

```
ReplaceAll[expr, u^3 -> u^expr]

ReplaceAll[expr, (_Symbol)^3 :> u^expr]
```

Note that it is also possible to use `Set` directly to manipulate a part of an expression and the expression itself.

```
expr[[2, 2]] = expr;
expr
```

We discuss this issue in detail in Subsection 6.3.3. `ReplacePart` replaces more than one subexpression. In this case, `ReplacePart` has to be called with four arguments.

> `ReplacePart[`*expression*`, `*newExpressionList*`, `*positionList*`, `*newExpressionPositionList*`]`
> replaces the element *expression*`[[`*positionList*`[[`*i*`]]]]` by the new expression *newExpression*`·`
> *List*`[[`*newExpressionPositionList*`[[`*i*`]]]]` for all *i*.

Here, the first, third, and sixth elements are replaced.

```
ReplacePart[{1, 2, 3, 4, 5, 6}, {11, 33, 66},
            {{1}, {3}, {6}}, {{1}, {2}, {3}}]
```

Look at the differences among the following three replacements. In the first case, nothing happens because the expression `a + b + c + d` does not match `Plus[]`. In the second case, because of the `Flat` and `OneIdentity` attribute of `Plus`, a "virtual" `Plus[]` is formed via `Plus[a, b, c, d]`$\longrightarrow$ `Plus[Plus[], Plus[a, b, c, d]]` that then allows us to apply the transformation rule and gives `a + b + c + d + e` as the result. In the last case, this rewriting happens repeatedly until the `MaxIterations` limit is exceeded.

```
Replace[a + b + c + d, HoldPattern[Plus[]] :> e]

a + b + c + d /. HoldPattern[Plus[]] :> e

a + b + c + d //. HoldPattern[Plus[]] :> e
```

The above was a practical introduction into patterns. Theoretical considerations concerning rules and rule applications, variable screening in rule applications, and its relation to the λ-calculus can be found in [29★].

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

■ 5.3.2 Large Numbers of Replacement Rules

To apply a large number of replacement rules in the most efficient way, we use `Dispatch`.

`Dispatch[`*rules*`]` produces an optimized list of the replacement rules in *rules*.

Here is a “larger” (but not more difficult) example that involves a long list.

```
table = Table[{i, j}, {i, 1, 25}]
```

The following replacement list is to be used on the individual elements.

```
ruleTab = Table[{i, j} -> i, {i, 1, 25}]
```

Here is the result of applying `ruleTab` to `table` (with a time measurement).

```
Timing[Do[table /. ruleTab, {10000}]]
```

Next, we look at an optimized form of the replacement rules.

```
ruleTabDispatch = Dispatch[ruleTab]
```

It is clearly faster.

```
Timing[Do[table /. ruleTabDispatch, {10000}]]
```

Doubling the length of the list shows the timing difference more pronounced.

```
table2 = Table[{i, j}, {i, 1, 2 25}];
{Timing[Do[table2 /. ruleTab, {10000/2}]],
 Timing[Do[table2 /. ruleTabDispatch, {10000/2}]]}
```

We do not discuss `FullForm`, the construction, and the operation of objects generated with `Dispatch`; see [67✱]. In Section 1.9.2 of the Symbolics volume [140✱] we will discuss a programming example which will make heavy use of `Dispatch`.

Be aware that only rules without explicit patterns (with `Blank[]`, …) can be dispatched and that otherwise no message is generated

```
Dispatch[Table[_ -> i, {i, 20}]]
```

For nondispatched rules, the time for applying them is proportional to the number of rules. The application time for dispatched rules is basically independent of the number of rules. The following inputs demonstrate this.

```
(* a list of numbers *)
tab = Table[If[EvenQ[j], I, j], {j, 2000}];

rules = Table[(_?(# === j&) -> 2j) /. j -> j, {j, 10^3}];
rulesD = Dispatch[Table[j -> 2j, {j, 10^3}]];

{Timing[tab /. rules;], Timing[Do[tab /. rulesD, {1000}]]}

(* double the number of rules *)
rules = Table[(_?(# === j&) -> 2j) /. j -> j, {j, 2 10^3}];
rulesD = Dispatch[Table[j -> 2j, {j, 2 10^3}]];

{Timing[tab /. rules;], Timing[Do[tab /. rulesD, {1000}]]}
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

5.3.3 Programming with Rules

In this subsection, we give a few typical examples of the efficient application of patterns and replacement rules. Some of the examples were winners in the programming contests held at *Mathematica* conferences. Most of the following implementations with patterns are very short and clear; but these program structures are generally not optimal with respect to speed, because the pattern matching requires a lot of work, as many more patterns than necessary may be tested to find the desired one (often pattern matching has exponential complexity). Thus, when efficiency is important, assign corresponding attributes to program the search for the desired structures, or use `Map` and similar functions to simultaneously process a larger number of expressions at once rather than one after another. (You should not draw the conclusion from these examples that programs with `ReplaceRepeatedBlankNullSequence` combination always win the programming contests at *Mathematica* conferences; using `FoldList` also has a good chance to win.)

RunEncode

The first example involves the so-called `RunEncode` problem [143✱]. Suppose we are given a list of positive integers: `{1, 1, 2, 3, 4, 4, 5, 3, 2, 2, 7, 7, 8, 8, 9, 9, 1, 1, 1}`.

Starting with this list, we want to compute a new list containing lists of the form {$number_i$, $numberOfNumber_i$} as elements. Here, $numberOfNumber_i$ gives the number of times that $number_i$ appears in a row. For the above list, this result would be `{{1, 2}, {2, 1}, {3, 1}, {4, 2}, {5, 1}, {3, 1}, {2, 2}, {7, 2}, {8, 2}, {9, 2}, {1, 3}}`.

In the first step, we convert every element of the given list to the form {*element*, 1}. Note the use of `ReplaceAll`.

```
{1, 1, 2, 3, 4, 4, 5, 3, 2, 2, 7, 7, 8, 8, 9, 9, 1, 1, 1} /.
                                          {a_Integer -> {a, 1}}
```

In the second step, we make use of the fact that …, {{$element_1$, $number_1$}, {$element_1$, $number_2$}, …} has to be replaced by …,{$element_1$, $number_1 + number_2$}, ….

Using `//.`, we do this until nothing more changes. The `a___` at the beginning and the `c___` at the end are needed because something may be there.

```
% //. {a___, {b_, i_}, {b_, j_}, c___} -> {a, {b, i + j}, c}
```

We now combine these processes.

```
RunEncode[list_List] := ((list /. {a_Integer -> {a, 1}}) //.
       {a___, {b_, i_}, {b_, j_}, c___} -> {a, {b, i + j}, c})
```

As the next example shows, this procedure works.

```
RunEncode[{1,
           2, 2,
           3, 3, 3,
           4, 4, 4, 4,
           5, 5, 5, 5, 5,
           6, 6, 6, 6, 6, 6,
           7, 7, 7, 7, 7, 7, 7,
           8, 8, 8, 8, 8, 8, 8, 8,
           9, 9, 9, 9, 9, 9, 9, 9, 9,
           8, 8, 8, 8, 8, 8, 8, 8,
           7, 7, 7, 7, 7, 7, 7,
           6, 6, 6, 6, 6, 6,
           5, 5, 5, 5, 5,
           4, 4, 4, 4,
           3, 3, 3,
           2, 2,
           1}]
```

To see in detail how *Mathematica* tries to match the pattern, we insert a `Print` command on the right-hand side with the new expression. Here, we do this process only in the second step because the first one is trivial.

```
(* auxiliary functions for formatting the print statements *)
toString[] = "_";
toString[s_] := ToString[s];
toString[s__] := StringJoin[ToString /@ {s}];

(({1, 1, 2, 2, 3, 3, 3, 4, 4, 4, 4} /.
                      {a_Integer -> {a, 1}}) //.
 ({a___, {b_, i_}, {b_, j_}, c___}) :>
             (Print[" a ⟶ " <> toString[a] <>
                    " b ⟶ " <> toString[b] <>
                    " c ⟶ " <> toString[c] <>
                    " i ⟶ " <> toString[i] <>
                    " j ⟶ " <> toString[j]]; {a, {b, i + j}, c}))
```

Although our implementation is already short, it can be made even shorter; see [149✱]. Using the function `Split` (to be discussed in the next chapter), we could implement the function `RunEncode` shortly and efficiently in the following manner.

```
RunEncode[list_List] := {First[#], Length[#]}& /@ Split[list]

RunEncode[{1, 1, 2, 2, 3, 3, 3, 4, 4, 4, 4}]
```

RulesToCycles

The so-called `RulesToCycles` problem [83✱] involves reordering a list of permutations into separate cycles. For example, consider the list `{1 -> 1, 2 -> 5, 5 -> 3, 3 -> 2, 4 -> 4, 7 -> 8, 9 -> 9, 8 -> 7}`.

Then the result should be `{{1}, {2, 5, 3}, {4}, {7, 8}, {9}}`.

In the first step, we rewrite all rules in lists.

```
{1 -> 1, 2 -> 5, 5 -> 3, 3 -> 2, 4 -> 4, 7 -> 8, 9 -> 9, 8 -> 7} /.
                                          (a_ -> b_) -> {a, b}
```

In the second step, we join all of the resulting sublists that belong together into larger sublists (here, we have to work with `ReplaceRepeated`, to find all possibilities). Note that new elements can be added at the beginning as well as at the end of the resulting lists.

```
% //. {{a___, {b__, c_}, d___, {c_, e__}, f___} -> {a, {b, c, e}, d, f},
       {a___, {b_, c__}, d___, {e__, b_}, f___} -> {a, {b, c}, d, f}}
```

In the third and last step, we remove each of the last arguments, because they now appear twice.

```
% /. {a_, b___, a_} -> {a, b}
```

Again, we combine the substitutions all into one routine.

```
RulesToCycles[l_List] := l /. (a_ -> b_) -> {a, b} //.
{{a___, {b__, c_}, d___, {c_, e___}, f___} -> {a, {b, c, e}, d, f},
 {a___, {b_, c__}, d___, {e__, b_}, f___} -> {a, {b, c}, d, f}} /.
                                          {a_, b___, a_} -> {a, b}
```

It works as expected.

```
RulesToCycles[{1 -> 2, 2 -> 3, 11 -> 11, 3 -> 4, 4 -> 1,
               9 -> 8, 8 -> 7, 7 -> 6, 6 -> 9}]
```

We could now go on and add a check for sensible input to `RulesToCycles`. A possibility of such a check is `Rules⋱ ToCycles[l:{__Rule} /; Sort[Map[First, l]] === Sort[Map[Last, l]]] :=...`; we will discuss the functions `Sort` and `Map` in the next chapter.

SortComplexNumbers

Next, we look at a sorting problem. (It could be easily solved with the command `Sort` to be discussed in the next chapter, but here we want to solve it using pattern-matching techniques.) The problem is to sort a list of complex numbers in increasing order according to their real parts, and for numbers with the same real part, in decreasing order according to their imaginary parts. Thus, `{2 + 5 I, 5, -8 I, -4 I, 4, 2 + 10 I}` should become `{-4I, -8 I, 2 + 10 I, 2 + 5 I, 4, 5}`.

In the first step, we sort according to increasing real parts.

```
{2 + 5 I, 5, -8 I, -4 I, 4, 2 + 10 I} //.
     {a___, b_, c___, d_, e___} :> {a, d, c, b, e} /; Re[d] < Re[b]
```

In the second step, we sort according to decreasing imaginary parts for numbers with the same real part.

```
% //. {a___, b_, c___, d_, e___} :>
      {a, d, c, b, e} /; (Re[b] == Re[d] && Im[d] > Im[b])
```

Note the parentheses used the replacement rules around `Condition[...]`.

```
Clear[beforehand, afterward, condition];
FullForm[beforehand :> afterward /; condition]
```

Thus, the condition is first checked for the right-hand side of the replacement rule. We again combine and test the resulting function.

```
SortComplexNumbers[l_List] := l //.
{a___, b_, c___, d_, e___} :> {a, d, c, b, e} /; Re[d] < Re[b] //.
{a___, b_, c___, d_, e___} :> {a, d, c, b, e} /;
                                 (Re[b] == Re[d] && Im[d] > Im[b])
```

```
SortComplexNumbers[{11 + I, 11 - I, 10, 9, 8, 7, 6 + I,
                    6 + 2 I, 6 + 3 I, 6 + 4 I}]
```

With a similar trick as above, we can again learn something about the "comparison strategy" used by *Mathematica*. On the right-hand side of `RuleDelayed`, we create lists `lre` and `lim` in which we store the pairs being compared by

Mathematica. The command `AppendTo` is discussed in the next chapter; it appends its second argument to its first, which is a symbol set to a list.

```
SortComplexNumbersWithInfo[l_List] :=
(lre = {}; lim = {}; l //.
{a___, b_, c___, d_, e___} :> {a, d, c, b, e} /;
                    (AppendTo[lre, {b, d}]; Re[d] < Re[b]) //.
    {a___, b_, c___, d_, e___} :> {a, d, c, b, e} /;
                    (AppendTo[lim, {b, d}];
                     Re[b] == Re[d] && Im[d] > Im[b]))
```

Note that in an expression of the form $expr_1$`;` $expr_2$`;` …`;` $expr_n$, only the last expression $expr_n$ is the result that is returned.

```
SortComplexNumbersWithInfo[{9, 8, 7, 6 + I, 6 + 2 I, 6 + 3 I}]
```

To sort the above five numbers into the desired order, a large number of comparisons are required.

```
lre // Short[#, 8]&
```

```
Length[%]
```

```
lim
```

```
Length[%]
```

Maxima

Here is the so-called maxima problem (proposed by R. Gaylord). Given a list of positive integers, make a new list of those numbers in the original list (in their original order) that are greater than all of their predecessors. Thus, for example, starting with the list `{3, 2, 8, 1, 10}`, `Maxima[{3, 2, 8, 1, 10}]` should give the result `{3, 8, 10}`.

The solution of this problem is very simple if we use `ReplaceRepeated` along with `BlankNullSequence`.

```
Maxima[l_List] := l //. ({a___, x_, y_, c___} /; y <= x) -> {a, x, c}
```

Here is an example.

```
Maxima[{1, 2, 3, 2, 1, 5, 3, 2, 8, 0, 0, 1, 23}]
```

Observe again the use of `Condition` in the first argument of `Rule`.

```
Unevaluated[({a___, x_, y_, c___} /; y <= x) -> {a, x, c}] // FullForm
```

Alternatively, the following code also works.

```
Maxima[l_List] := l //. {a___, x_, y_, c___} :> ({a, x, c} /; y <= x)
```

```
Maxima[{1, 2, 3, 2, 1, 5, 3, 2, 8, 0, 0, 1, 23}]
```

Splitting

The split problem [103★] involves dividing a given list of objects into smaller lists whose lengths are prescribed by a second list. For example, given the list of objects `{a, b, c, 1, 2, 3, {}, {{}}}` and the list of lengths `{3, 0, 3, 2}`, the result of `Splitting[{a, b, c, 1, 2, 3, {}, {{}}}, {3, 0, 3, 2}]` should be `{{a, b, c}, {}, {1, 2, 3}, {{}, {{}}}}`. That is, a 0 in the list of lengths corresponds to an empty set. First, we program the construction of one step in the computation of the new list. For this reason, we combine the list to be constructed, the list to be divided, and the list of lengths together into one new list, and then apply the following replacement rule.

```
{{a1___}, {a2___, b2___}, {a3_, b3___}} :>
          {{a1, {a2}}, {b2}, {b3}} /; Length[{a2}] == a3
```

Applying this rule takes `a3` elements from the list `{a2…}` to be divided, and adds them at the end of the list `{a1…}` to be constructed. In addition, these elements are removed from the second list along with the corresponding number in the third list. We now look at two steps of how this process works in our example.

```
{{}, {a, b, c, 1, 2, 3, {}, {{}}}, {3, 0, 3, 2}} /.
        {{a1___}, {a2___, b2___}, {a3_, b3___}} :>
                   {{a1, {a2}}, {b2}, {b3}} /; Length[{a2}] == a3
```

```
% /. {{a1___}, {a2___, b2___}, {a3_, b3___}} :>
          {{a1, {a2}}, {b2}, {b3}} /; Length[{a2}] == a3
```

Using `ReplaceRepeated`, we repeat this process until it stops naturally.

```
{{}, {a, b, c, 1, 2, 3, {}, {{}}}, {3, 0, 3, 2}} //.
      {{a1___}, {a2___, b2___}, {a3_, b3___}} :>
            {{a1, {a2}}, {b2}, {b3}} /; Length[{a2}] == a3
```

It remains only to remove the empty lists in the second and third places. (Here, we could of course use `Part[…, 1]` instead of applying a rule for doing this job.)

```
% /. {l_, {}, {}} -> l
```

Combining the above steps, we get the following program.

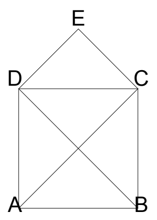
```
Splitting[list_, s_] := ({{}, list, s} //. {{a1___}, {a2___, b2___},
                        {a3_, b3___}} :> {{a1, {a2}}, {b2}, {b3}} /;
                            Length[{a2}] == a3) /. {l_, {}, {}} -> l
```

We give one final example.

```
Splitting[{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, E^E},
          {0, 0, 0, 1, 2, 2, 3, 3, 4, 1, 0, 0}]
```

House of the Nikolaus

How many possibilities exist to draw the little house below in one stroke starting from the point A and not traversing any line twice? (The graphic comes from the German children's rhyme "Das—ist—das—Haus—vom—Ni—ko—laus".)



Our drawing stroke will be of the form `Line[` *stroke*$_1$`,` *stroke*$_2$, …`]`. The used strokes and the unused strokes we will collect in a list `{`*alreadyDrawn*`,` *stillNotDrawn*`}`. The next stroke to be drawn must start at the ending point of the last stroke. The following rules implement this property. By using the pattern `{u___, y_, v___}`, we can use unoriented *stillNotDrawn* line segments.

```
addLineRule = {Line[begin___, {x__, y_}],
                {α___, Line[{u___, y_, v___}], γ___}} :>
                {Line[begin, {x, y}, {y, u, v}] , {α, γ}};
```

We have three possibilities to start from the point A. Either the stroke AB or the stroke AC or the stroke AD. Let us have a look at the house starting with the stroke AD.

```
startConfiguration1 = {Line[{a, d}],
 {Line[{a, b}], Line[{a, c}], Line[{d, c}], Line[{b, c}],
  Line[{b, d}], Line[{d, e}], Line[{e, c}]}};
```

Applying the rule `addLineRule` one time results in the double stroke ADC.

```
startConfiguration1 /. addLineRule
```

Because we are interested in all possible ways to draw the little house, we use `ReplaceList`. For the second stroke (starting from the point D), we have three possibilities.

```
 ReplaceList[startConfiguration1, addLineRule]
```

Now, we just repeat the application of the rule `addLineRule` with a `ReplaceList` until all line segments are drawn. The following `Nest` implements this process.

```
Nest[(# /. {Line[a___], b_} :>
     ReplaceList[{Line[a], b}, addLineRule])&, startConfiguration1,
     (* use all remaining seven line segments *)  7]
```

Removing the unnecessary list brackets from the last results gives the following 16 possibilities to draw the house when starting with the stroke AD.

```
res1 = Level[%, {-3}] //. (* remove {{}} *) {{}} :> Sequence[]

Length[res1]
```

In a similar way, we can now calculate the 12 possible ways to start with the stroke AB.

```
startConfiguration2 =
{Line[{a, b}],
 {Line[{a, d}], Line[{a, c}], Line[{d, c}], Line[{b, c}],
  Line[{b, d}], Line[{d, e}], Line[{e, c}]}};

res2 = Nest[(# /. {Line[a___], b_} :>
                  ReplaceList[{Line[a], b}, addLineRule])&,
            startConfiguration2, 7] // Level[#, {-3}]&
```

And finally, we have again 16 possibilities to start with the stroke AC.

```
startConfiguration3 =
{Line[{a, c}],
 {Line[{a, d}], Line[{a, b}], Line[{d, c}], Line[{b, c}],
  Line[{b, d}], Line[{d, e}], Line[{e, c}]}};
```

```
res3 = (Nest[(# /. {Line[a___], b_} :>
                 ReplaceList[{Line[a], b}, addLineRule])&,
           startConfiguration3, 7] // Level[#, {-3}]&) //.
                                        {{}} :> Sequence[]
```

So, we end up with 44 different possibilities.

```
allPossibilities =
(* unite the three lists *)
First[{{}, {res1, res2, res3}} //.
      {{l___}, {α___, {a___, b_Line, c___}, β___}} :>
      {{l, b}, {α, {a, c}, β}}] (* eliminate doubles *) //.
   {α___, l_, β___, l_, γ___} :> {α, l, β, γ};
```

```
Length[allPossibilities]
```

Interestingly, the last stroke always ends at the point B.

```
allPossibilities[[All, -1, -1]]
```

To see the 44 different possible ways to draw the house of the Nikolaus, we color the stroke continuously as it goes on (we start with red and end with red). The function `drawColoredHouse` implements this process.

```
drawColoredHouse[Line[l__]] :=
Block[{a = {0, 0}, b = {1, 0}, c = {1, 1},
       d = {0, 1}, e = {1/2, 3/2}, n = 30},
     MapIndexed[{Hue[#2[[1]]/(8n)], Line[#1]}&,
    Partition[ Flatten[Table[#[[1]] + k/n(#[[2]] - #[[1]]),
              {k, 0, n}]& /@ {l}, 1], 2, 1]]]
```

Here are the 44 different houses.

```
Show[GraphicsArray[Partition[
Graphics[drawColoredHouse[#], PlotRange -> All,
         AspectRatio -> Automatic]& /@ allPossibilities, 11]]]
```

At the end of this subsection, let us once again stress that the use of patterns and replacement rules is a very convenient way to treat complicated patterns. For simple iterative problems, `Nest`-like constructions are typically much faster. Let us study one example, so-called polypaths [27✱], [45✱]. The idea is to take a trapezoidal quadrilateral and repeatedly fold it along its diagonal. A polypath is then the set of the edges of the quadrilateral depending on the number of foldings. We describe the coordinates of the polygon vertices $\{x, y\}$ and use complex numbers of the form $x + i\, y$ for compact notation.

Let this be our starting quadrilateral.

```
start = {0.45965 I, 1.00624 + 0.53158 I,
         1.00624 - 0.53158 I, -0.45965 I};
```

The process of folding means to transform (p, q, r, s) into $q, r, s, (s-q)\,\overline{(p-q)/(s-q)} + q$ (see [27✱] for details). Here is a replacement rule that does this procedure repeatedly.

```
pointRule = {a___, b:{p_, q_, r_, s_}} :>
({a, b, {q, r, s, Conjugate[(p - q)/(s - q)](s - q) + q}} /;
                                       Length[{a}] < 1000);
```

Now let us do 1000 foldings and measure the time used by `ReplaceRepeated`.

```
({start} //. pointRule); // Timing
```

The next approach we use is a recursive function definition.

```
Clear[f]

f[{a___, b:{p_, q_, r_, s_}}] :=
      (f[{a, b, {q, r, s, Conjugate[(p - q)/(s - q)](s - q) + q}}] /;
                                                   Length[{a}] < 1000);

f[l_?(Length[#] > 1000&)] = l;
```

The last stopping rule is only applied when the first rule does not match. We see this fact by looking at the ordering of the above two definitions in `DownValues`.

```
??f
```

Here again is the time needed to do 1000 foldings.

```
f[{start}]; // Timing
```

The next approach we study here is the definition of a function `f` that just folds one time, and this definition is used repeatedly by `NestList`. Now, we can use an easier pattern; exactly one element has to be matched every time.

```
f[{p_, q_, r_, s_}] := {q, r, s, Conjugate[(p - q)/(s - q)](s - q) + q};
```

This approach is much faster.

```
NestList[f, start, 1000]; // Timing
```

The last method of folding is conceptually the same as the others, but now we use a pure function instead of `f`.

```
NestList[{#2, #3, #4, Conjugate[(#1 - #2)/(#4 - #2)]
          (#4 - #2) + #2}&[Sequence @@ #]&, start,
         1000]; // Timing
```

Using the command `Apply` (to be discussed in the next chapter), the last variant can be slightly shortened.

```
NestList[{#2, #3, #4, Conjugate[(#1 - #2)/(#4 - #2)]
          (#4 - #2) + #2}& @@ #&, start, 1000]; // Timing
```

Using compilation (to be discussed in detail in Chapter 1 of the Numerics volume [139✱]), the last variant can still be made faster by about a factor of 10.

```
cf = Compile[{{s, _Complex, 1}, {n, _Integer}},
             NestList[{#[[2]], #[[3]], #[[4]],
                       Conjugate[(#[[1]] - #[[2]])/(#[[4]] - #[[2]])]
              (#[[4]] - #[[2]]) + #[[2]]}&, s, n]]

Timing[(* do 100 times *) Do[cf[start, 1000], {100}]]
```

Here is an example of how the folding looks after 10000 turns.

```
Show[Graphics[{PointSize[0.004], Map[Point[{Re[#], Im[#]}]&,
     cf[start, 10000], {2}]}], AspectRatio -> Automatic]
```

Depending on the starting polygon, polypaths can show an unexpected variety of shapes. The following graphics show some of the possible shapes.

```
With[{o = 2 10^4},
Show[GraphicsArray[Apply[Graphics[{PointSize[0.004],
       MapIndexed[{Hue[#2[[1]]/o], #1}&, Map[Point[{Re[#], Im[#]}]&,
                  cf[{#1, #2, Conjugate[#2], -#1}, o], {2}]]},
             AspectRatio -> 1, PlotRange -> All]&, #, {1}]]]& /@
 Partition[(* polygon data *)
{{0.827238 I, 0.904941 + 0.605458 I}, {0.684092 I, 0.336662 + 0.054269 I},
 {0.240041 I, 0.362625 + 0.983340 I}, {0.354225 I, 0.808103 + 0.310474 I},
 {0.807148 I, 0.802408 + 0.163400 I}, {0.245298 I, 0.924847 + 0.198034 I},
 {0.252427 I, 0.866921 + 0.743293 I}, {0.263133 I, 0.942035 + 0.209090 I},
 {0.379324 I, 0.966640 + 0.411363 I}}, 3]]
```

After having discussed patterns and replacement rules, we will relax for a minute and enjoy a little animation. We take a parametrized polygon with vertices $\{A,\ x\,B,\ x\,\overline{B},\ \overline{A}\}$ and visualize the polypath as a function of x. The lengths of the x-ranges of the animations are between 0.15 and 0.016.

```
picture[x_, color_, points_:1000] :=
Graphics[{PointSize[0.003], color, Map[Point[{Re[#], Im[#]}]&,
          cf[{0.25649382714 I, x (0.4429289741158 + 0.1591829440563 I),
              x (0.44292897411 - 0.1591829440563 I), -0.256493827140 I},
             points], {2}]}, AspectRatio -> 1, PlotRange -> All]
```

```
With[{frames = 5},
Do[Show[GraphicsArray[
     {{picture[0.840 + k/frames 0.1500, Hue[0.00], 10000],
       picture[0.862 + k/frames 0.0060, Hue[0.12], 10000],
       picture[0.949 + k/frames 0.0020, Hue[0.22], 10000],
       picture[0.962 + k/frames 0.0016, Hue[0.76], 10000]}}]],
   {k, 0, frames}]]
```

```
With[{frames = 90},
Do[Show[GraphicsArray[
     {{picture[0.840 + k/frames 0.1500, Hue[0.00], 10000],
       picture[0.862 + k/frames 0.0060, Hue[0.12], 10000]},
      {picture[0.949 + k/frames 0.0020, Hue[0.22], 10000],
       picture[0.962 + k/frames 0.0016, Hue[0.76], 10000]}}]],
   {k, 0, frames}]]
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

5.4 String Patterns

In the last sections, we discussed in detail pattern-matching related issues for expressions. Frequently one has to analyze, transform, and build strings whose elements do not correspond to *Mathematica* expressions. Most of the pattern matching related functions discussed in this chapter have an equivalent for strings. We will discuss the most important string matching functions here; for a complete discussion, see the Advanced Documentation in the help browser.

A string in *Mathematica* is enclosed in quotes. Patterns are not part of a string, but of a `StringExpression`.

`StringExpression[`*stringsAndPatterns*`]`
represents a string for pattern matching purposes.

Here is a simple example. The first element of the string expression `se1` is the string `"1"` and the second is the integer `2`. In output, string expressions are displayed in infix form using `~~`. This means a string expression is a symbolic concatenation of explicit strings and patterns representing strings (that potentially have restrictions on their form).

```
se1 = StringExpression["1", 2]
```

If consecutive elements in a string expression are strings, then they are automatically concatenated to one string. Here is an example of this situation.

```
StringExpression["1", "2", 3] // InputForm
```

In analogy to the meaning of `Blank*[]` for expression patterns, the patterns _, __, and ___ in a string expression stand for one characters, a sequence of one or more characters, and a sequence of zero or more characters. Also, their names equivalents like *x*`:`_, repetitions *pattern*`..`, and *pattern*`...` are recognized in string expressions as the corresponding patterns.

We now have a look at all built-in functions having the name `String`... and having nonstring analog. Here is a list of these functions.

```
stringFunctions = ToExpression /@
 Select[Names["String*"], MemberQ[Names["*"], StringDrop[#, 6]]&]
```

To find out if a given string matches a given pattern, the function `StringMatchQ` can be used.

`StringMatchQ[`*string*`,` *stringExpressionOrString*`]`
gives `True` if the string expression matches the pattern represented by *stringExpressionOr*-*String* and `False` otherwise.

Here are two examples. The string `"123456789"` starts with `"1"`, ends with `"9"`, and has one or more characters in between.

```
StringMatchQ["123456789", "1" ~~ __ ~~ "9"]
```

The same string does not match the pattern `"1" ~~ _ ~~ _ ~~ "9"` because it allows for two in-between characters only.

```
StringMatchQ["123456789", "1" ~~ _ ~~ _ ~~ "9"]
```

Here are some possibilities to match any string of length two or larger.

```
StringMatchQ["abc", StringExpression[__]]

StringMatchQ["abc", StringExpression[_ ..]]

StringMatchQ["abc", StringExpression[_ | __]]

StringMatchQ["abc", StringExpression[_ | __]]
```

The second `*Q` function that operates on strings is `StringFreeQ`.

`StringFreeQ[`*string*`,` *stringExpressionOrString*`]`
gives `True` if the string expression does not contain the pattern represented by *stringExpres*-*sionOrString* and `False` otherwise.

The substring "ab" does not appear in the following string.

```
StringFreeQ["this string is free of what?", "ab"]
```

The position of a given substring can be determined with the function `StringPosition`.

`StringPosition[`*string*, *stringExpressionOrString*`]`

returns the character positions of realizations of the string pattern *stringExpressionOrString* in the string *string*.

The function `StringPosition` returns a list of lists. The sublist indicates the character positions of the first and the last characters that match.

```
StringPosition["12345678987654321", "7"]
```

The next input determines the string position of a substring that begins with the substring "1" and is followed by at least one more character.

```
StringPosition["12345678987654321", "12" ~~ __]
```

The last call to `StringPosition` returned the whole string—the longest possible match. To obtain the shortest possible match, we can use the function `ShortestMatch`.

`ShortestMatch[`*stringExpressionOrString*`]`

represents in string matching functions the shortest match for the string pattern *stringExpres*⁄ *sionOrString* (in the string).

`LongestMatch[`*stringExpressionOrString*`]`

represents in string matching functions the longest match for the string pattern *stringExpression OrString* (in the string).

Now, we obtain the positions of the first three characters—the shortest possible match.

```
StringPosition["12345678987654321", ShortestMatch["12" ~~ __]]
```

Many of the string-analyzing and -manipulating functions have options. Here are all of the options.

```
Union[Flatten[(First /@ Options[#])& /@ stringFunctions]]
```

For many string-matching operations, the most important of these options is `Overlaps`. This option can be set to `True`, `False`, and `All`. In the first case, one overlap between successive matches is possible, in the second none, and for the `All` option setting all possible string pattern realizations are taken into account.

This means, that for the following example of a character followed by one or more character, followed by another character, we have 5, 15, and 120 possible matches.

```
StringPosition["12345678987654321", ShortestMatch[_ ~~ __ ~~ _],
               Overlaps -> False]

StringPosition["12345678987654321", ShortestMatch[_ ~~ __ ~~ _],
               Overlaps -> True]

StringPosition["12345678987654321", ShortestMatch[_ ~~ __ ~~ _],
               Overlaps -> All]
```

Next, we use the function `Import` to load the Amazon web pages for this book from Amazon Germany and Amazon France (in the URL, the book is identified by the ISBN number).

```
(* import German page *)
imD = Import["http://www.amazon.de/exec/obidos/ASIN/0387942823", "Text"];

(* import French page *)
imF = Import["http://www.amazon.fr/exec/obidos/ASIN/0387942823", "Text"];

(*
(* import British page *)
 imGB = Import["http://www.amazon.co.uk/exec/obidos/ASIN/0387942823", "Text"];
*)
```

We locate for the shortest phrases of the form "Preis … EUR" and "Notre prix … EUR" or similar.

```
{(* German phrase *)
 Select[StringPosition[imD, ShortestMatch["EUR"]],
        (Abs[Subtract @@ #] < 50)&, 1],
 (* French phrase *)
 Select[StringPosition[imF, ShortestMatch["EUR"]],
        (Abs[Subtract @@ #] < 50)&, 1]}
```

Here are the extracted phrases. We see some HTML formatting and the price of the book.

```
{StringTake[imD, %[[1, 1]] + {-6, 6}],
 StringTake[imF, %[[2, 1]] + {-6, 6}]}
```

We can count substrings using the function `StringCount`.

`StringCount[`*string*`,` *stringExpressionOrString*`]`

returns the number of occurrences of realizations of the string pattern *stringExpressionOr*`: `*String* in the string *string*.

Here is a long string of the digits 1 to 9.

```
longString[n_] := longString[n] =
  StringJoin @ Table[ToString[IntegerPart[(9 Abs[Sin[k]])] + 1], {k, n}];
```

Here are the first and last digits of this string for $n = 10^4$.

```
Short[longString[10^4], 12]
```

Here we count how often the digits 1 to 9 occur in the string `longString`.

```
Function[n, {n, StringCount[longString[10^4], n]}, {Listable}][
                        {"1", "2", "3", "4", "5", "6", "7", "8", "9"}]

Function[n, {n, StringCount[longString[10^4], n ~~ __ ~~ n,
                           Overlaps -> All]}, {Listable}][
                        {"1", "2", "3", "4", "5", "6", "7", "8", "9"}]
```

The equivalent function to `Replace` for strings is `StringReplace`.

`StringReplace[`*string*`,` *replacementRule*`]`

replaces the substrings matching the first argument of *replacementRule* with its second argument in the string *string*.

In the next input, we replace all occurrences of any character followed by the character `"1"` with two copies of these two characters.

```
StringReplace["a1b1c1d1e1f1", ξ:(_ ~~ "1") :> (ξ <> ξ)]
```

In the next input, the first character must be "a" or "f".

```
StringReplace["a1b1c1d1e1f1", ξ:(("a" | "f") ~~ "1") :> (ξ <> ξ)]
```

The next input imports the web page with additional materials from the *GuideBook*'s home page.

```
guideBookAdditionPage =
     Import["http://mathematicaguidebooks.org/additions.shtml", "Text"];

Short[guideBookAdditionPage, 16]
```

And the next input extracts the titles of all downloadable notebooks.

```
(* remove the html formatting *)
Function[s, StringReplace[s, {"title\"> " -> "", "\n" -> ""}],
        {Listable}][(* extract text lines with titles *)
                    StringCases[guideBookAdditionPage,
                                ShortestMatch["title\">" ~~ __ ~~ "\n"]]]
```

StringReplace does carry out one possible replacement and returns the new string. A list of all strings that one can obtain through a specified replacement is returned by StringReplaceList.

StringReplaceList[*string*, *replacementRule*]
gives a list of all strings that can be obtained by applying the rule to the string *string*.

There are 55 possibilities to replace one or more consecutive characters in the 9-character string "123456789".

```
StringReplaceList["123456789", x:___ :> "X"]
```

Using a longer string and carrying out the same replacement as in the last input, gives (of course) more potential matches.

```
StringReplaceList[longString[100], x:___ :> "X"] // Length
```

We end with a small application. The following input generates a list of all the main chapter notebooks of T*he Mathematica Book* (excluding the reference guide and the index) that are visible in the help browser.

```
(* all files in the Mathematica directory *)
allMathematicaBookFiles =
   Select[FileNames["*", $InstallationDirectory, Infinity],
          (StringMatchQ[#, "*MainBook*"] &&
           MatchQ[StringTake[#, {-7, -6}], "0_" | "1_" | "2_" | "3_"])&];

(* number of files *)
λ = Length[allMathematicaBookFiles]
```

By removing all formatting information (for font changes), deleting all graphics, and by replacing all mathematical typesetting by the string "$\mathcal{FORMULA}$", we obtain one string of the whole book.

```
mathematicaBookText =
Module[{nb, text},
StringJoin @ Flatten[Table[
  nb = Get[allMathematicaBookFiles[[k]]];
  text = Flatten[
  Flatten[nb[[1]] //. Cell[CellGroupData[l__, _], ___] :> l] //.
               StyleBox[c_, _] :> c //.
               Cell[TextData[l_], ___] :> l //.
               BoxData[f_] :> "𝓕𝓞𝓡𝓜𝓤𝓛𝓐" //. ButtonBox[c_, __] :> c //.
               OutputFormData[c_] :> c //.
               _GraphicsData :> {} //. Cell[c_, __] :> c], {k, λ}]]];
```

The resulting string has more than one million characters.

```
μ = StringLength[mathematicaBookText]
```

The next input counts the number of occurrences of three strings.

```
Function[word, {word, StringCount[mathematicaBookText, word]},
         Listable][{"The " | " the ", "Mathematica ", " set up "}]
```

We convert all upper-case letters and to lower case letters.

```
mathematicaBookTextLC = ToLowerCase[mathematicaBookText];
```

Next, we determine the positions $p_k(letter)$ ($k = 1, 2, \ldots, q(letter)$) of the of the 26 letters "a", "b", …, "z".

```
allLCLetters = Characters["abcdefghijklmnopqrstuvwxyz"]

letterPositions =
Function[letter, (First /@ #)& @
             StringPosition[mathematicaBookTextLC, letter],
           {Listable}] @ allLCLetters;
```

The next graphic shows $p_k(letter)/\mu - k/q(letter)$ as a function of $p_k(letter)$. We see quite visible deviations from the average value 1.

```
Show[Graphics[{Thickness[0.002],
       Table[{Hue[(k - 1)/32],
             Line[MapIndexed[{#1, #1/μ -
                              #2[[1]]/Length[letterPositions[[k]]]}&,
                             letterPositions[[k]]]]}, {k, 26}]}],
     PlotRange -> All, Frame -> True];
```

We end with determining the frequency of all pairs of letters.

```
doubleLetterCounts =
Map[StringCount[mathematicaBookTextLC, #]&,
    Outer[StringJoin, allLCLetters, allLCLetters], {2}];
```

Here are the resulting counts of letter pairs.

```
ListPlot3D[doubleLetterCounts,
           Ticks -> {MapIndexed[{#2[[1]], #1}&, allLCLetters],
                     MapIndexed[{#2[[1]], #1}&, allLCLetters], Automatic},
           PlotRange -> All]
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

Overview

```
Get[ToFileName[ReplacePart[
          "FileName" /. NotebookInformation[EvaluationNotebook[]],
          "ChapterOverview.m", 2]]];

ChapterOverview["Programming", 5]
```

Exercises

1.^{L1} `myExpand`

Write a function `myExpand` using `Rule`, which multiplies out polynomials and products.

2.^{L1} `ReplaceAll` versus `ReplaceRepeated`

Discuss the following replacements:

```
replacement = {x + 1 -> px};

1 + x + 1/(1 + x) + (1 + x)^(1 + x) + f[1 + x] /. replacement

Plus[1 + x, 1/(1 + x), (1 + x)^(1 + x), f[1 + x]] /. replacement

1 + x + 1/(1 + x) + (1 + x)^(1 + x) + f[1 + x] //. replacement

plus[1 + x, 1/(1 + x), (1 + x)^(1 + x), f[1 + x]] /. replacement
```

Here, the function `plus` should have the same attributes as the function `Plus`.

3.^{L1} All Other Patterns with `s, t, _, _, :`

Examine all of the ways of creating a syntactically correct *Mathematica* expression from `s, t, _, _,` or `s, t, _, _, :` using at most two blanks. From the 1440 possible combinations, about two-thirds as many syntactically correct expressions exists, which reduce to about 8% different ones. An implementation of a program producing them is given in the solution (its operation will become clear after the discussion in Chapter 6).

4.^{L1} $\cos(x)^n \to f(\sin(x))$

Consider the following sum:

$$\cos(x)^2 + \cos(x)^4 + \cos(x)^6 + \cos(x)^8 + \cos(x)^{10} + \cos(x)^{12} + \cos(x)^{14} + \cos(x)^{16}$$

Express this sum using only $\sin(x)^i$. Use a rule-based approach.

5.^{L1} `a[a]`

Examine the results of the following *Mathematica* inputs, and explain what happens.

a) `Clear[a]; a = a`

b) `Clear[a]; a := a; a`

c) `Clear[a]; a[a_] = a; a[a]`

d) `Clear[a]; a[a_] := a; a[a]`

e) `Clear[a]; a = a == a; a`

f) `Clear[a]; a := a == a; a`

g) `Clear[a]; a := a == a; Unevaluated[a]`

h) `Clear[a]; a := a == a; Hold[a]`

i) `Clear[a]; a := Unevaluated[a] == Unevaluated[a]; a`

j) `Clear[a]; a := Unevaluated[a] == a; a`

k)
```
Remove[a, x, y, z, Aah]
a/: b[x_][a] := Null /; (Clear[a]; b[y_][a][z_] = Aah[y, z]; False)
b[a][a][a]
```

6.^{L1} Extended `Equal`

Modify the built-in function `Equal` so that equations can be added, multiplied, and raised to given powers. In addition, make it possible to add something to both sides or multiply both sides of an equation by a constant.

7.^{L2} Weights for Finite Differences

Finite difference methods [73✱] of higher order provide an important alternative to the finite element method. To use them, we need corresponding weights. For the one-dimensional case, the following recurrence formulas hold for the weights $c_{i,j}^k$ of the nodes x_j ($j = 0, 1, \ldots, n$) in the approximation of a kth derivative with a total of $i+1$ nodes. Here, x_0 is the point at which the derivative is to be approximated:

$$f_{(i)}^{(k)}(x)\Big|_{x=x_0} \approx \sum_{j=0}^{i+1} c_{i,j}^k\, f(x_j)$$

$$\begin{aligned}
c_{i,j}^k &= \frac{1}{x_i - x_j}\left(x_i\, c_{i-1,j}^k - k\, c_{i-1,j}^{k-1}\right), \quad j = 0, 1, \ldots, i-1 \\
c_{i,i}^k &= \frac{\omega_{i-2}(x_{i-1})}{\omega_{i-1}(x_i)}\left(k\, c_{i-1,i-1}^{k-1} - x_{i-1}\, c_{i-1,i-1}^k\right) \\
\omega_i(x) &= \prod_{j=0}^{i}\left(x - x_j\right)
\end{aligned}$$

(The order of the other nodes x_j is arbitrary.)

Find the associated initial conditions for these recurrence formulas, and implement the computation of the $c_{i,j}^k$. (For the derivation of these recurrence formulas, see [49✱], [50✱], [133✱], [51✱], [134✱], [11✱], [125✱], [148✱], and [34✱].)

Use this finite difference approximation to calculate reliable values for the first 20 derivatives of $\hat{\zeta}(1/2)$. Here, $\hat{\zeta}(s)$ is $\hat{\zeta}(s) = s\,(s-1)\,\pi^{-s/2}\,\Gamma(s/2)\,\zeta(s)/2$ [92✱], [18✱], [107✱] and fulfills the functional equation $\hat{\zeta}(s) = \hat{\zeta}(1-s)$. In the last equation, $\Gamma(s)$ is the Gamma function (in *Mathematica* `Gamma[s]`) and $\zeta(s)$ is the Riemann Zeta function (in *Mathematica* `Zeta[s]`). What is remarkable about these derivatives?

8.^{L3} Operator Product, *q*-, *h*-Binomial Theorem, Ordered Derivative

a) Define a function `operatorProduct` describing the noncommutative, associative multiplication of operators. Suppose the operators are given in the form `○[operatorIndex]`. All quantities that do not depend on the operators (numbers, constants, variables) should be factored out (before computing the operator product). Implement the additivity and associativity and a way to multiply out positive integer powers of sums of operators. If the reader has an appropri ate application of such operator products, implement it also.

b) The famous binomial theorem $(x+y)^n = \sum_{k=0}^{n} \binom{n}{k} y^k x^{n-k}$ has two very interesting generalizations for noncommuting x and y. In the case of $x\,y = q\,y\,x$ ($q \in \mathbb{C}$) [68✱], [17✱], the binomial theorem becomes the q-binomial theorem [131✱], [46✱], [6✱], [130✱], [77✱], [79✱], [14✱], [122✱], [89✱], [3✱], [135✱], [31✱]

$$(x+y)^n = \sum_{k=0}^{n} \begin{bmatrix} n \\ k \end{bmatrix}_q y^k x^{n-k}$$

$$\begin{bmatrix} n \\ k \end{bmatrix}_q = \frac{(q;q)_n}{(q;q)_k\,(q;q)_{n-k}}$$

$$(a;q)_n = \prod_{k=0}^{n-1} \left(1 - a\,q^k\right),\ a \in \mathbb{C},\ n \in \mathbb{N}.$$

How often do the transformation rules in the transformation of $(x+y)^{10}$ to expanded form get applied?

In the case of $x\,y = y\,x + h\,y^2$ ($h \in \mathbb{C}$), the generalization of the binomial theorem is the h-binomial theorem [12✱], [63✱]

$$(x+y)^n = \sum_{k=0}^{n} \binom{n}{k} h^k \left(\frac{1}{h}\right)_k y^k x^{n-k}$$

$$(a)_n = \prod_{k=0}^{n-1} (a+k),\ a \in \mathbb{C},\ n \in \mathbb{N}.$$

For $1 \le n \le 8$, verify explicitly the q-binomial theorem and the h-binomial theorem by straightforward calculation. (For the q-h-binomial theorem, see [13✱], and for other generalizations, see [97✱].)

The q-binomial coefficients $\left[\begin{smallmatrix} n \\ k \end{smallmatrix}\right]_q$ appear, for instance, when q-differentiating q-functions. If $\frac{d_q f(x)}{d_q x}$ is the q-derivative of a function $f(x)$ defined by [78✱], [42✱], [69✱], [41✱], [20✱], [82✱]

$$\frac{d_q f(x)}{d_q x} = \frac{f(x) - f(q\,x)}{(1-q)\,x}$$

(this derivative can be interpreted as a discrete derivative approximation after a change of variables [40✱]) and $\frac{d_q^n f(x)}{d_q x^n}$ the nth q-derivative ($\frac{d_q^1 f(x)}{d_q x^1} = \frac{d_q f(x)}{d_q x}$)

$$\frac{d_q^n\, f(x)}{d_q\, x^n} = \frac{d_q\, \frac{d_q^{n-1}\, f(x)}{d_q\, x^{n-1}}}{d_q\, x}$$

then the following two identities hold:

$$\frac{d_q^n\, f(x)\, g(x)}{d_q\, x^n} = \sum_{k=0}^{n} \begin{bmatrix} n \\ k \end{bmatrix}_q \frac{d_q^{n-k}\, f\left(q^k\, x\right)}{d_q\, x^{n-k}}\, \frac{d_q^k\, g(x)}{d_q\, x^k}$$

$$\frac{d_q^n\, f(x)}{d_q\, x^n} = \frac{1}{(1-q)^n\, x^n} \sum_{k=0}^{n} (-1)^k \begin{bmatrix} n \\ k \end{bmatrix}_q q^{-(k-1)\,k/2-(n-k)\,k}\, f\left(x\, q^k\right)$$

Check these two identities for $0 \leq n \leq 10$.

c) In [118✱] an "ordered derivative" of an operator product $\hat{x}_{\alpha_1}\, \hat{x}_{\alpha_2} \ldots \hat{x}_{\alpha_n}$ with respect to a sequence of corresponding classical symbol $x_{\beta_1} \ldots x_{\beta_m}$ has been defined. The $\hat{x}_\alpha$ are assumed to be noncommuting and the x_α to be commuting. The "ordered derivative" δ *operatorProduct* / δ *classicalSymbols* is defined in the following way:

$$\frac{\delta\, \hat{x}_\alpha}{\delta\, x_\beta} = \begin{cases} 1 \text{ if } \alpha = \beta \\ 0 \text{ else} \end{cases}$$

$$\frac{\delta\, \hat{x}_\alpha}{\delta\left(x_{\beta_1} \cdots x_{\beta_m}\right)} = \prod_{k=1}^{m} \frac{\delta\, \hat{x}_\alpha}{\delta\, x_{\beta_k}}$$

$$\frac{\delta\left(\hat{x}_{\alpha_1} \cdots \hat{x}_{\alpha_n}\right)}{\delta\left(x_{\beta_1} \cdots x_{\beta_m}\right)} = \sum_{k=0}^{m} \frac{\delta\left(\hat{x}_{\alpha_1} \cdots \hat{x}_{\alpha_l}\right)}{\delta\left(x_{\beta_1} \cdots x_{\beta_k}\right)}\, \frac{\delta\left(\hat{x}_{\alpha_{l+1}} \cdots \hat{x}_{\alpha_n}\right)}{\delta\left(x_{\beta_{k+1}} \cdots x_{\beta_m}\right)}$$

where l in the last definition is an arbitrary integer between 1 and $n-1$ (the results of the "ordered derivative" does not depend on l). Implement a function that carries out the "ordered derivative".

The $\mathbb{f}$ in the "ordered derivative" of

$$\frac{\delta\left(\hat{x}_{\alpha_1} \cdots \hat{x}_{\alpha_n}\right)}{\delta\left(x_{\beta_1} \cdots x_{\beta_m}\right)} = \mathbb{f}\ productOfThex_\alpha s$$

(*productOfThex* is proportional to the "ordinary derivative" $\partial\left(x_{\alpha_1} \cdots x_{\alpha_n}\right) / \partial\left(x_{\beta_1} \cdots x_{\beta_m}\right)$ with all products of the same symbol collapsed) counts how many possibilities exist to delete the string of $x_{\beta_1} \cdots x_{\beta_m}$ from the string $x_{\alpha_1} \cdots x_{\alpha_n}$. (The x_{β_1} must appear in order, but not contiguously in $x_{\alpha_1} \cdots x_{\alpha_n}$.) Check this statement for

$$\frac{\delta\left(\hat{x}_1\, \hat{x}_2^2\, \hat{x}_3^3\, \hat{x}_4^4\, \hat{x}_5^5\, \hat{x}_6^4\, \hat{x}_7^3\, \hat{x}_8^2\, \hat{x}_9\right)}{\delta\left(x_1\, x_2\, x_3\, x_4\, x_5\, x_6\, x_7\, x_8\, x_9\right)}.$$

d) Let $\mathbf{A}(t)$ be a parametrized, nonsingular $n \times n$ matrix. Using $\partial\left(\mathbf{A}(t).\mathbf{A}(t)^{(-1)} - \mathbf{1}\right) / \partial t = \mathbf{0}$ ($\mathbf{0}$ being the n-dimensional matrix with all elements being 0 and $\mathbf{1}$ being the n-dimensional identity matrix) can derive the expression $\partial \mathbf{A}(t)^{(-1)} / \partial t = -\mathbf{A}(t)^{(-1)}.\, (\partial \mathbf{A}(t) / \partial t).\mathbf{A}(t)^{(-1)}$ for the derivative of the inverse matrix $\mathbf{A}(t)^{(-1)}$ (here differentiation with respect to t is understood componentwise). Calculate the explicit form of $\partial^5 \left(\mathbf{A}(t)^{(-1)}\right)^5 / \partial t^5$. Simplify the result when all occurring matrices commute. Count how often the needed definitions are applied during the calculation.

9.^{L2} Patterns and Replacements

Program solutions to the following problems; base the programs on pattern matching and the use of replacement rules.

a) Given a list of elements (some of which may appear more than once), construct a list containing all (different) permutations of the elements.

b) Given a list of the form `{`*integer*`,` *nZeros*`}`, for example, `{23, 0, 0, 0, 0, 0}`, construct all lists of integers a_i ($i = 1, \ldots, n+1$) (i.e., of the same length as the original list) with $\sum_{i=1}^{n+1} a_i = integer$ (i.e., which have the same sum as the original list). Put the a_i in increasing order: $a_i \leq a_{i+1}$, ($i = 1, \ldots, n-1$).

c) Given lists of the form `{0, …, 0,` *number*$_1$`, 0, …0,` *number*$_2$`, …,` *number*$_n$`, 0, …0}` and $\{newNumber_1, \ldots, newNumber_n\}$, construct a new list from the first list by replacing $number_i$ by $newNumber_i$ ($i = 1, n$) $\{a_1, a_2, \ldots, a_{n-1}, a_n\}$. (This problem was proposed by R. Gaylord.).

d) Given a list of positive integers, construct a list containing all pairs of numbers with no common factor.

e) Given a list of positive integers in decreasing order, construct the Ferrer conjugate of this list. The Ferrer conjugate is defined in the following way [142★], [33★], [56★], and [4★]: Associate with the list `{`n_1`,` n_2`, …,` n_k`}` an array of dots; n_1 in the first row, n_2 in the second, and so on. Then, the list of the lengths of the columns, starting from the left, is the Ferrer conjugate. An example: the Ferrer conjugate of `{5, 3, 2, 1}` is `{4, 3, 2, 1, 1}` as can be seen by

```
• • • • •
• • •
• •
•
```

10.^{L1} Hermite Polynomials, Peakons

a) The Hermite polynomials $H_n(x)$ satisfy the following identity: $x\, H_n(x) = H_{n+1}(x)/2 + n\, H_{n-1}(x)$, $n \in \mathbb{N}$.

Program the repeated use of this identity in terms of the form $x^m\, H_n(x)$ to write them as linear combinations of the Hermite polynomials without x-dependent prefactors. Make the program work for user-defined objects `H[n, z]`. (Do not modify the built-in function `HermiteH`.)

b) Show that $\psi(x, t) = c \exp(-|x - c\, t|)$ is a solution of the nonlinear Camasso–Holm partial differential equation [24★], [25★], [86★], [87★], [64★], [48★], [85★], [60★], [36★], [2★], [37★], [88★]

$$\frac{\partial \psi(x,t)}{\partial t} - \frac{\partial^3 \psi(x,t)}{\partial x^3} + 3\,\psi(x,t)\,\frac{\partial \psi(x,t)}{\partial x} = 2\,\frac{\partial \psi(x,t)}{\partial x}\,\frac{\partial^2 \psi(x,t)}{\partial x^2} + \frac{\partial^3 \psi(x,t)}{\partial x^2\,\partial t} + \psi(x,t)\,\frac{\partial^3 \psi(x,t)}{\partial x^3}.$$

11.^{L1} `f[x___] :=` …

What outputs correspond to the following inputs?

a)
```
f[x___] := x + 1
f[]
f[1, 2, 3]
```

b)
```
f[x___] := x - 1
f[]
f[1, 2, 3]
```

c)
```
f[x___] := Subtract[x, 1]
f[]
```

```
  f[1, 2, 3]
```

d) `f /: HoldPattern[HoldPattern[Verbatim[HoldPattern[f]]]] := 4`

```
  HoldPattern[f]
```

12.^{L1} Result and Error Messages

Predict the output and warning/ messages generated when evaluating the following:

```
{1, 2} //. {{x___, y___} :> ({x, Unique[c], y} /;
                                (Head[{x}[[-1]]] =!= Symbol &&
                                 Head[{y}[[ 1]]] =!= Symbol))}
```

13.^{L1} Patterns

a) Construct at least five different patterns that match integers greater than 1 and less than 9.

b) To obtain the result `{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11}`, identify which arguments must be given to `f`, defined by

```
f[Condition[Condition_, Condition; True],
  Optional[Blank_, Optional],
  Pattern[Pattern, Blank[Integer]],
  Four:(4 | 4.),
  PatternTest[Pattern[PatternTest, Blank[]], PatternTest; True&],
  Alternatives:Alternatives[Alternatives, 6],
  Flat_Flat,
  Stub:Blank[Orderless[OneIdentity]],
  HoldPattern_HoldPattern?(# === #&),
  HoldPattern[Set[3, 4]]] :=
      {Condition, Blank, Pattern, Four, PatternTest,
       Alternatives, Flat[[1]], Stub[[1]], 9, HoldPattern[[1]]}
```

c) What are the results of the following inputs?

```
Remove[a]

SetAttributes[a, HoldAll]

f:a[a_] := Function[#, Hold[#], {HoldAll}][f]&[Unique[a]]

{a[a], a[b], a[2a], a[a + a]}
Remove[a]

SetAttributes[a, HoldAll]

f:a[a_] := Function[#, Hold[#], {HoldAll}][f]&[Unique["a"]]

{a[a], a[b], a[2a], a[a + a]}
```

d) What are the results of the following inputs?

```
SetAttributes[AtomQ, HoldAll]
{AtomQ[1/2], AtomQ[1 + I]}
```

e) What is the result of the following input?

```
blank[Pattern[Blank, Blank[Blank]]] = Blank

blank[Blank[Blank]]
```

f) Predict the results and side effects of the following three inputs.

```
f1[x0_] := Block[{x = x0}, Print[C1]; x = x + 1; Print[C2] /; Positive[x]]

f1[-2]

f2[x0_] := Block[{x = x0}, ToExpression[
                "Print[C1]; x = x + 1; Print[C2] /; Positive[x]"]]

f2[-2]

f3[x0_] := Block[{x = x0}, (Print[C1]; x = x + 1;
                           condition[Print[C2], Positive[x]]) /.
                                             condition -> Condition]

f3[-2]
```

14.^{L1} Replacements

Explain what happens when evaluating the following expressions:

a) `{1, 2, 3, 4, 5} //. {a__, b_, c_, d___} :>`
```
                              If[b > 2, {b, c, d}, {a, b, c, d}]
```

b) `{1, 2, 3, 4, 5} //. {a___, b_, c_, d___} :>`
```
                              If[b > 2, {b, c, d}, {a, b, c, d}]
```

c) `{1, 2, 3, 4, 5} //. {a__, b_, c_, d___} :> {b, c, d} /; b > 2`

d) `{1, 2, 3, 4, 5} //. {a___, b_, c_, d___} :> {b, c, d} /; b > 2`

e) `{1, 2, 3, 4, 5} //. (({a__, b_, c_, d___} /; b > 2) :> {b, c, d}/; b > 2)`

f) `{1, 2, 3, 4, 5} //. (({a___, b_, c_, d___} /; b > 2) :> {b, c, d}/; b > 2)`

15.^{L1} Puzzles

a) What might have been the In[1] in the following example?

```
In[2]:= a
Out[2]= True

In[3]:= And[a, a]
Out[3]= False
```

Give at least three different possibilities for In[1]. Find some solutions that do not involve unprotecting built-in functions.

b) Predict the result of the following input:

```
(Im[3 I] == 0) // Function[{x}, Block[{I}, x], {HoldAll}]
```

c) Predict the result of the following input:

```
Hold[With[{z = Abort[]}, z^2]] /. z_?Quit :> Quit[]
```

d) Have a look at:

```
On[]; 2/3 === Unevaluated[2/3]
```

We see there the line:

```
                2     2
SameQ::trace: - === - --> False.
                3     3
```

Explain this “surprising” printout!

e) For which built-in symbols *builtInSymbol* does *builtInSymbol* == *builtInSymbol* not yield `True`? Why?

f) What will be the result of the following input?

```
(𝕏[_?(# === _?#0&), C_ /; MatchQ[C, _ /; MatchQ[C, _]]] := 𝕐;
 𝕏[_?(# === _?#0&), C_ /; MatchQ[C, _ /; MatchQ[C, _]]])
```

g) For many cases, `IntegerQ[`*x*`]` will return `True` or `False`. Find three different values for *x* such that something else is returned.

h) Evaluating `f[a, b]` after making the function definition

```
SetAttributes[f, {Flat, OneIdentity}]

f[ξ_] := ξ
```

leads to iteration errors. How can one change the definition `f[ξ_] := ξ` to prevent this problem?

i) Let *g*, *h*, and *i* in the following be all possible combinations of `HoldPattern` and `Verbatim`. For which of the eight possible combinations of *g*, *h*, and *i* does the input

```
f[g[h][x_]] := x; f[i[1]]
```

return 1?

j) After defining `f` by

```
With[{a = x}, HoldPattern[f[y_, g[y_] = y^2]] := a]
```

find arguments, such that the definition for `f` will be used.

k) Predict the result of the following input.

```
Block[{Function}, (#&[2]) /. Function -> Print]
```

l) Predict the side effects of evaluating the following:

```
SetAttributes[{𝓜, TagUnset, ToString}, HoldAllComplete]

𝓜[e_] := (e /: HoldPattern[e:h_[___, e, ___]] :=
          (Print["Found: ", h, " ", HoldForm[e]];
           ToExpression[# <> " /: HoldPattern[e:h_[___, " <>
                            # <> ", ___]] =."]&[ToString[e]];
           𝓜[h, e]; e))

𝓜[h_, e_] := (h /: HoldPattern[e:ℓ_[___, e, ___]] :=
             (Print["Found: ", HoldForm[e]];
              TagUnset @@ {h, UpValues[h][[1, 1, 1]]}; 𝓜[ℓ, e]; e))
```

```
𝓜[x];
α[1, β[y], a[b[c[2, d[f[x], 1]]]]]
```

m) A bivariate function $f(x, y)$ can be written in separated form $f(x, y) = \varphi(x)\,\phi(y)$ in a neighborhood of a point $\{x_0, y_0\}$ if $f(x_0, y_0) \neq 0$ and [123✱], [94✱], [145✱], [120✱], [98✱]

$$f(x, y)\,\frac{\partial^2 f(x, y)}{\partial x\,\partial y} = \frac{\partial f(x, y)}{\partial x}\,\frac{\partial f(x, y)}{\partial y}.$$

What is “wrong” with the following function `separableVariablesQ` that checks if a function f of the two variables x and y can be written in separated form?

```
separableVariablesQ[f_, {x_, y_}, {x0_, y0_}] :=
 (Simplify[f /. {x -> x0, y -> y0}] =!= 0) &&
  Simplify[f D[f, x, y] - D[f, x] D[f, y]] === 0
```

n) Predict the result of the following input.

```
SetAttributes[PrimeQ, HoldAll]

PrimeQ[2 + 3 I, {GaussianIntegers -> True}]
```

o) What might have been the In[1] in the following example? (No unprotecting of built-in symbols was involved.)

```
In[2]:= {NumericQ[%], NumberQ[%], MemberQ[%, _?InexactNumberQ],
        StringLength[StringDrop[ToString[
                                DownValues[In][[$Line - 1]]], 22]],
        Context /@ Cases[%, _Symbol, {-1}, Heads -> True]}
Out[2]= {True, False, True, 9, {System`}}
```

p) Construct an example of expressions `a` and `b` such that `FreeQ[a, b]` yields `False` and `Position[a, b]` yields `{}`.

16.^{L1} Evaluation Sequence

Discuss the evaluation sequence in the following four examples:

```
(f[x_] := g) /; c

(f[x_] /; c) := g

(f[x_] := g /; c)

(f[x_ /; c] := g)
```

17.^{L1} Nested Scoping

Predict the results of the following inputs.

a) `Clear[f]; f[x_] := Function[x, x]; f[y]`

b) `With[{x = z},Function[x, x]]`

c) `Function[x, x] /. x -> z`

d) `Clear[f]; Function[x, f[x_] := x^2][y]; DownValues[f]`

e) `Clear[f]; With[{x = y}, f[x_] := x^2]; DownValues[f]`

f) `Function[y, Function[x, x + y]][x]`

g) `Clear[f]; f[y_] := Function[x, x + y]; f[x]`

h)
```
Clear[f];
Module[{x, y, z = a}, f[x, y_, z] := Function[x, x + y + z]];
DownValues[f]
```

i)
```
Clear[f];
With[{z = a},
Module[{x, y}, f[x, y_, z] := Function[x, x + y + z]]];
DownValues[f]
```

18.^{L1} Why `{b,b}`?

Explain why it might be possible to get the following behavior. (For reproducing this behavior, the reader might have to redo the `Table[a, {10000}] // Union` line a few times until one get the result shown here.)

```
In[1]:= a := b /; EvenQ[Last[Date[]]]

In[2]:= Table[a, {10000}] // Union
Out[2]= {b, b}
```

Solutions

1. `myExpand`

Here is a possible solution.

```
myExpand[expression_?PolynomialQ] :=
expression //. {(* expand powers *)
                (a_ + b_)^c_Integer?(# > 1&) ->
                 a (a + b)^(c - 1) + b(a + b)^(c - 1),
                (* expand products *)
                (a_ + b_.)(c_ + d_) -> a c + b c + a d + b d}
```

Note the use of only one blank in the patterns. Because `Plus` has the attribute `Flat`, expressions of the form `(a + b + c + d + … + p)^c` are nevertheless recognized. Here is a simple example.

```
myExpand[(1 + 2x) (3x + 4x)^2 (1 - (2 x + 3)^2)^2]
```

Here are four more examples.

```
myExpand[(3 + 5u) (6 + 9r)] == Expand[(3 + 5u) (6 + 9r)]

myExpand[(2 + 4x + 6x^2)^3] == Expand[(2 + 4x + 6x^2)^3]

(* or shorter, in one line:
  myExpand[#] == Expand[#]&[((3 + 5u) (6 + 9r))^3 (a + h)^4] *)
myExpand[((3 + 5u) (6 + 9r))^3 (a + h)^4] ==
  Expand[((3 + 5u) (6 + 9r))^3 (a + h)^4]

myExpand[1 + (1 + (1 + (1 + ξ^2)^2)^2)^2] ==
  Expand[1 + (1 + (1 + (1 + ξ^2)^2)^2)^2]
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

2. `ReplaceAll` versus `ReplaceRepeated`

Here is the replacement rule.

```
replacement = {x + 1 -> px}
```

Despite `ReplaceAll`, "only" the first summand is replaced (we might expect five instances of `px` in the result).

```
1 + x + 1/(1 + x) + (1 + x)^(1 + x) + f[1 + x] /. replacement
```

Here is the same thing written out.

```
Plus[1 + x, 1/(1 + x), (1 + x)^(1 + x), f[1 + x]] /. replacement
```

Here is another `plus`, without attributes. Now, everything is replaced with `ReplaceAll`.

```
plus[1 + x, 1/(1 + x), (1 + x)^(1 + x), f[1 + x]] /. replacement
```

Even with the attributes of `Plus`, the two functions `plus` and `Plus` do not behave in the same way.

```
Attributes[Plus]
```

```
SetAttributes[plus, {Flat, Listable, NumericFunction,
                     OneIdentity, Orderless}];
plus[1 + x, 1/(1 + x), (1 + x)^(1 + x), f[1 + x]] /. replacement
```

The reason that the replacement did not work is the internal structure of the following structure.

```
1 + x + 1/(1 + x) + (1 + x)^(1 + x) + f[1 + x] // FullForm
```

The first `x` and the first 1 do not "belong together". Combining them and replacing them is the job of `ReplaceAll`. In *f*`[Plus[1, x]]`, the subexpression `1 + x` forms one unit from the beginning. With `ReplaceRepeated`, everything is replaced.

```
1 + x + 1/(1 + x) + (1 + x)^(1 + x) + f[1 + x] //. replacement
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

3. All Other Patterns with `s, t, _, _, :`

If this code is evaluated, we get a list of lists, each with three elements. In the inner lists, the first component contains (in a list) the different orderings of `s, t, _, _, :`, the second component contains the *Mathematica* expression, and its `FullForm` is in the third component. Most of the following constructions do not make much sense, but they are all syntactically correct.

```
allPatterns =
Module[{allInputStrings, syntacticallyCorrectInputs, fullForms},
(* form all permutations *)
allInputStrings = StringJoin @@@ Union[
   Permutations[{"s", " ", "_", "t", " ", "_"}],
   Permutations[{"s", " ", ":", "_", "t", " ", "_"}]];
(* select syntactically correct inputs *)
syntacticallyCorrectInputs = Select[allInputStrings, SyntaxQ];
(* generate full form of inputs *)
fullForms = Sort[{ToString[FullForm[ToExpression[#]]], #}& /@
                                     syntacticallyCorrectInputs];
(* group equivalent inputs *)
{#[[1, 1]], Last /@ #}& /@ Split[fullForms, #1[[1]] === #2[[1]]&]];
```

```
Short[allPatterns, 16]
```

Thus, we see that the 936 different inputs that make sense reduce to 117 different ones.

```
{Length[allPatterns],
 allPatterns /. {_, l_List} :> Length[l] /. List -> Plus}
```

For a mathematical analysis of all programs of a given size, see [23✱].

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

4. $\cos(x)^n \to f(\sin(x))$

This transformation can be implemented, for instance, using replacements.

```
Sum[Cos[x]^i, {i, 0, 16, 2}] /.
          {Cos[x]^i_?EvenQ :> Expand[(1 - Sin[x]^2)^(i/2)]}
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

5. a[a]

a) Here, `a` is immediately assigned the value `a` via `Set`. The result of this assignment is, of course, `a`.

```
Clear[a]; a = a
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

b) Here, `a` is assigned the value `a` via `SetDelayed`. The result of the later computation "of `a` as `a`" is, of course, `a`.

```
Clear[a]; a := a;  a
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

c) This assignment is uncommon, but correct. The function `a[`*something*`]` is immediately assigned its argument as its value. A later call of the function `a[a]` returns its argument, namely, `a`.

```
Clear[a]; a[a_] = a; a[a]
```

```
a[x]
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

d) This example is the analogous construction with `SetDelayed`. The function `a[`*something*`]` is assigned the value of its argument. A later computation of the function `a[a]` returns its argument as `a`. This case is very similar to the analogous `Set` construction.

```
Clear[a]; a[a_] := a; a[a]
```

The `a` in `a_` is a pattern, so we can call `a` with any argument and it will evaluate to the argument.

```
a[x]
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

e) The variable `a` is immediately assigned the value of the expression `Equal[a, a]`, that is, `True`. A later call on the variable `a` returns the value of `a`, namely, `True`.

```
Clear[a]; a = a == a; a
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

f) This is the analogous construction with `SetDelayed`. The truth value of `Equal[a, a]` is first computed with the call of `a`. Now, the problems start. With the call of `a`, `a` is replaced by `a == a`. Then, the following attempt to determine the truth value of this assertion causes the `$RecursionLimit` to be exceeded, because to compute `a`, it must be replaced by `a == a`, and to compute these `a`s, and so on. We constrain the running time of the following recursive calculation.

```
TimeConstrained[Clear[a]; a := a == a; a, 2]
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

g) This case is especially tricky. In spite of `Unevaluated`, `$IterationLimit` is exceeded. To see why, we look at the same input, but with `Unevaluated[a]` as a separate input.

```
Clear[a];
a := a == a;
Unevaluated[a]
```

We now recognize the problem. `Unevaluated` is an argument in
`CompoundExpression[Clear[a], a := a == a, Unevaluated[a]]}.`

(See also Exercise 2 in Chapter 4). `Unevaluated` vanishes, and the computation of `a` leads to the same problems as before. Again, the following code has to be aborted.

```
TimeConstrained[Clear[a]; a := a == a; Unevaluated[a]; , 2]
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

h) `Hold` is safe in this regard. It definitely prevents the computation.

```
Clear[a]; a := a == a; Hold[a]
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

i) Here, we do not get an infinite loop. The two occurrences of `Unevaluated` in `Equal` prevent the recursive computation of `a`, and `Equal` immediately takes effect.

```
Clear[a]; a := Unevaluated[a] == Unevaluated[a]; a
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

j) One `Unevaluated` does not suffice; *Mathematica* attempts to compute the `a` on the right, which again causes an infinite loop. But here we do not need to intervene manually. We apply `Short` to avoid getting `Unevaluated` 255 times.

```
Clear[a];
Short[a := Unevaluated[a] == a; a, 8]
```

```
Count[%, Unevaluated, {-1}, Heads -> True]
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

k) In the first step, a definition is made for the pattern of the `b[`*something*`][a]`.

```
a /: b[x_][a] := Null /; (Clear[a]; b[y_][a][z_] = Aah[y, z]; False)
```

```
??a
```

When this expression is encountered, *Mathematica* evaluates the right-hand side. The right-hand side contains a condition (which always returns `False`) to be tested. In the process of testing the condition, the definition for `a` is cleared and a definition for `b`, for a pattern of the form `b[`*something*$_1$`][a][`*something*$_2$`]`, is installed. So in the input `b[a][a][a]`, the head `b[a][a]` triggers the rule for `a`, which installs the rule for `b` and returns the input `b[a][a][a]` unevaluated because the `Condition` returns `False`. The rule for `b` matches the input `b[a][a][a]`, which evaluates to `Aah[a, a]`.

```
b[a][a][a]
```

a has no definition at the moment.

```
??a
```

But b does have a definition.

```
??b
```

Using Trace, we see the intermediate steps.

```
Remove[a, b, x, y, z, Aah]

a/: b[x_][a] := Null /; (Clear[a]; b[y_][a][z_] = Aah[y, z]; False)

Trace[b[a][a][a]]
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

6. Extended Equal

Here is a possible modification of the built-in function Equal. The first of the two new rules relates to the manipulation of two equations, whereas the second rule applies to one equation. Note that we associate all rules with Equal via TagSet.

```
Unprotect[Equal];

(* add, multiply, … two equations *)
f_[u_ == v_, x_ == y_] ^= (f[u, x] == f[v, y]);

(* apply a function (with possible parameters) to an equation *)
f_[param1___, x_ == y_, param2___] ^=
          f[param1, x, param2] == f[param1, y, param2];

Protect[Equal];
```

Here are a few examples of the operation of our “new” Equal.

```
(a1 == a2)^(b1 == b2)

(a1 == a2) * (b1 == b2)

(a1 == a2) + (b1 == b2)

σ + (a1 == a2)

σ  (a1 == a2)

Sin[a1 == a2]

𝔽[a1 == a2]

𝕗[x, z11 == z12, y]
```

These extensions to Equal are often useful, especially for working interactively. We will not need them further.

```
Unprotect[Equal];
Clear[Equal];
Protect[Equal];
```

See also [117✱] for extending the capabilities of Equal.

Here is another possibility. The function ApplyOperationsToEquations applies each of the operations from the list *operations* to the list of equations *equations*. (We use the construction Flatten[{*equations*}] to allow *equations* to be a single equation instead of a List. And we use `HoldPattern[Equal[args___]]` to avoid that `Equal[args___]` evaluates to `True`.)

```
ApplyOperationsToEquations[operations_, equations_] :=
 Flatten[Function[eq,
   Function[op, eq /. HoldPattern[Equal[args___]] :>
              (Equal @@ (op /@ {args}))] /@ Flatten[{operations}]] /@
                                        Flatten[{equations}]]
```

Here is an example.

```
ApplyOperationsToEquations[{Re, Im, Abs},
                           {a + I b == c + I d == e + I f,
                            Sin[α + I β] == Cosh[γ + I δ]}] //
          (* separate real and imaginary parts *) ComplexExpand
```

```
Σ (* session summary *)  TMGBs`PrintSessionSummary[]
```

7. Weights for Finite Differences

The missing initial conditions are $c_{0,0}^{0} = 1$, $c_{i,j}^{k} = 0$ for $k < 0$, and $c_{i,j}^{k} = 0$ for $i < k$.

This recursion leads to the implementation below. (We encapsulate the computation somewhat and, for the sake of efficiency, save some of the intermediate values.)

```
FiniteDifferenceWeights[ord_Integer?(# >= 0&), node_List] :=
Module[{x, y, c, ω},
       Evaluate[Table[x[i], {i, 0, Length[node] - 1}]] = node;
       (* the function ω *)
       ω[i_, y_] := ω[i, y] = Product[y - x[j], {j, 0, i}];
       (* initial condition for c *)
       c[0, 0, 0] = 1;
       (* recursion for c *)
       c[k_?(# >= 0&), i_, j_] := (c[k, i, j] = 0) /; i < k;
       c[k_?(# <  0&), i_, j_] = 0;
       c[k_?(# >= 0&), i_, i_] := (c[k, i, i] =
                    ω[i - 2, x[i - 1]]/ω[i - 1, x[i]] *
                    (k c[k - 1, i - 1, i - 1] -
                    x[i - 1] c[k, i - 1, i - 1])) /; i >= k;
       c[k_?(# >= 0&), i_, j_] := (c[k, i, j] = 1/(x[i] - x[j])*
                    (x[i] c[k, i - 1, j] - k c[k - 1, i - 1, j])) /;
                                                (i >= k && j <= i);
       (* the weights *)
       Table[c[ord, Length[node] - 1, j],
                   {j, 0, Length[node] - 1}]] /; ord < Length[node]
```

We now look at a few of the resulting weights. Here are symmetric approximations for the second derivatives using 2 *iMax* + 1 nodes with a spacing of 1.

```
Table[FiniteDifferenceWeights[2, Table[i, {i, -iMax - 1, iMax + 1, 1}]],
      {iMax, 0, 3}]
```

Here is a visualization of the weights for the node numbers 3, 5, …, 41. The central nodes are always weighted most. The left graphic shows the weights directly and the right graphic shows the logarithm of the absolute values of the weights.

```
Show[GraphicsArray[
Block[{n = 20, $DisplayFunction = Identity},
Function[f, Graphics[Reverse[
MapIndexed[{Hue[#2[[1]]/26], Line[#1]}&,
           MapIndexed[{#2[[2]] - #2[[1]] - 1, f[#1]}&,
  (* the finite differences *)
  Table[FiniteDifferenceWeights[2, Range[-iMax - 1, iMax + 1]],
      {iMax, 0, n}], {2}]]], PlotRange -> All, Frame -> True]] /@
 (* show weights and log(abs(weights)) *) {Identity, Log[Abs[#]]&}]]]
```

For the first derivative and equidistant nodes, we get coefficients that can be expressed through factorials [57✱], [58✱], [109✱].

```
Table[FiniteDifferenceWeights[1, Table[i, {i, -iMax - 1, iMax + 1, 1}]],
      {iMax, 0, 5}]

Table[Table[If[k === 0, 0, (-1)^(k - 1) n!^2/(k (n + k)! (n - k)!)],
            {k, -n, n}], {n, 6}]
```

Here are some left-sided approximations for first derivatives using *iMax* + 1 nodes with a spacing of 1.

```
Table[FiniteDifferenceWeights[1, Table[i, {i, 0, iMax}]],
      {iMax, 1, 8}]
```

The corresponding expressions are also computed for symbolic arguments.

```
FiniteDifferenceWeights[4, {x0, x1, x2, x3, x4}] // Simplify
```

We now examine the quality of these approximations of the second derivative of cos(x) at $x = 0$ as a function of the number of nodes, where the nodal coordinates are

$$\{-0.1, 0, +0.1\}$$
$$\{-0.2, -0.1, 0, +0.1, +0.2\}$$
$$\{-0.3, -0.2, -0.1, 0, +0.1, +0.2, +0.3\}$$
$$\{-0.4, -0.3, -0.2, -0.1, 0, +0.1, +0.2, +0.3, +0.4\}$$
$$\{-0.5, -0.4, -0.3, -0.2, -0.1, 0, +0.1, +0.2, +0.3, +0.4, +0.5\}$$
$$\ldots$$

In the following application, we use the fact that `Cos` has the attribute `Listable`. (The command `.` (`Dot`) is discussed in the next chapter.)

```
Table[1 + (Cos[Table[i/10, {i, -iMax - 1, iMax + 1, 1}]]).
      FiniteDifferenceWeights[2,
        Table[i/10, {i, -iMax - 1, iMax + 1, 1}]],
                    {iMax, 0, 12}] // N[#, 30]& // N
```

For a compact one-liner to derive these finite difference weights, see [52✱]; for the derivation for higher dimensional finite difference formulas in *Mathematica*, see [61✱]; for the calculation of general finite difference formulas, see [1✱]. For the perfect discretizations of differential operators in general, see [62✱], [72✱], [28✱], [65✱], and [5✱].

Now, we will deal with the second part of the question, which is the function $\hat{\zeta}(s) = s\,(s-1)\,\pi^{-s/2}\,\Gamma(s/2)\,\zeta(s)/2$.

```
ξ[s_] := s (s - 1) Pi^(-s/2) Gamma[s/2] Zeta[s]/2
```

Because of the functional equation $\hat{\zeta}(s) = \hat{\zeta}(1-s)$, all odd derivatives vanish identically. A plot shows the form of the function near $\hat{\zeta}(1/2)$.

```
Plot[ξ[s] - ξ[1/2], {s, 0.4999, 0.5001}, PlotRange -> All]
```

Using the function `FiniteDifferenceWeights`, we calculate a function `dApprox` that gives the approximative value of $\hat{\zeta}^{(o)}(1/2)$. To calculate this approximation, we use $2n+3$ symmetric s-values around $s = 1/2$. To make sure that we can control rounding errors, we carry out all calculations with precision *prec*.

```
dApprox[o_, {n_, δ_}, prec_] :=
Module[{t = Table[i δ, {i, -n - 1, n + 1, 1}], fdws, ξValues, sum},
          (* the finite difference weights *)
          fdws = FiniteDifferenceWeights[o, t];
          (* the function values *)
          ξValues = N[ξ[1/2 + t], prec];
          (* the approximation for the derivative *)
          sum = 0;
          Do[sum = sum + fdws[[k]] ξValues[[k]], {k, 2n + 3}];
          sum]
```

As expected, the odd-order derivatives vanish.

```
dApprox[1, {3, 1/100}, 30]
```

```
dApprox[3, {3, 1/100}, 30]
```

For the second derivative, the value approaches 0.022971….

```
Table[dApprox[2, {n, 1/100}, 30], {n, 3, 5}]
```

To get reliable values for the higher derivatives, we implement an increasing number of s-values around $s = 1/2$ until we have about five reliable digits. The function `gooddApprox` returns the value of the derivative as well as the number of s-values needed to achieve the required precision.

```
gooddApprox[o_, d_, δ_, prec_] :=
Module[{n = 2 o + 1, oldDerivative = 10, newDerivative},
(* until the approximation is precise enough *)
While[newDerivative = dApprox[o, {n, δ}, prec];
       Abs[Abs[(oldDerivative - newDerivative)/newDerivative]] > 10^-d,
       n = n + 1;
       oldDerivative = newDerivative];
       {newDerivative, n}]
```

```
Do[Print[{2o, Date[], gooddApprox[2o, 5, 1/1000, 100] // N}], {o, 10}]
```

The interesting fact about these derivatives is that they seem to be all positive. The statement that they are all positive [105✱], [106✱] is equivalent to the famous Riemann hypothesis [39✱], [43✱], [138✱], [66✱], [70✱]. The similar statement that for all positive integer n the quantities $\partial^n\left(s^{n-1}\log(\zeta(s))\right)/\partial s^n\,|_{s=1}$ are positive [84✱], [91✱], [30✱] is also equivalent to the Riemann hypothesis.

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

8. Operator Product, *q*-, *h*-Binomial Theorem, Ordered Derivative

a) Here is an implementation of the various properties. The head $\mathcal{OP}$ indicates an operator product.

```
(* Associativity *)
OP[a___, OP[b___], c___] := OP[a, b, c]
(* Additivity *)
OP[a___, b1_ + b2_, c___] := OP[a, b1, c] + OP[a, b2, c]
(* o[i] - independent expressions are not operators *)
OP[x_?(FreeQ[#, o[_]]&)] := x
(* factor out o[i] - independent factors *)
OP[a___, x_?(FreeQ[#, o[_]]&), b___] := x OP[a, b]
OP[a___, x__?(FreeQ[#, o[_]]&) y_, b___] := x OP[a, y, b]
(* multiply out powers of sums *)
OP[a___, b_Plus^n_Integer?(# > 1&), c___] :=
                      OP[a, Table[b, {n}] /. List -> Sequence, c]
(* to reduce the notation, write products of
   operators as powers (this may not always be desirable) *)
OP[a___, o[index_]^n1_., o[index_]^n2_., c___] :=
OP[a, o[index]^(n1 + n2), c]
(* single operators are not operator products *)
OP[op[index_]] = o[index];
```

We now look at a few examples of how this definition works.

■ Use additivity

```
OP[1 + o[t] + o[z]^2]
```

■ Remove factors

```
OP[2, o[t], o[z]^2, r, o[t]]
```

```
OP[2 o[t], o[z]^2, r o[t], 67 z o[g]]
```

■ Multiply out powers

```
OP[(s + o[k])^2]
```

```
OP[2 + o[t], o[z]^3, op[z], 1]
```

■ Use associativity

```
OP[OP[o[1], o[2]^2], OP[o[2]^3, o[3]]]
```

Now, we sketch an application: the Campbell-Baker-Hausdorff formula (see [116✱], [115✱], [112✱], [137✱], [96✱], [136✱], [141✱], [54✱], [38✱], [35✱], [132✱], [75✱], [126✱], [100✱], [80✱], [128✱], [111✱], [19✱], [81✱], [8✱], [113✱], [71✱], [9✱] and [119✱] for details). Let λ and μ be two noncommuting operators. Then, for σ in

$$\begin{aligned} e^{\lambda}\, e^{\mu} &= e^{\sigma} \\ \sigma &= \ln\left(e^{\lambda}\, e^{\mu}\right) = \lambda + \int_0^1 \Psi(\exp(\mathrm{Ad}(\lambda))\exp(t\,\mathrm{Ad}(\mu)))\,\mu\,dt \\ \Psi(z) &= \frac{z}{z-1}\ln(z), \end{aligned}$$

and in the superoperator $\mathrm{Ad}(\zeta)$

$$\mathrm{Ad}(\zeta)\,\eta = [\zeta, \eta] \equiv \zeta\eta - \zeta\eta.$$

Expanding $\Psi(\zeta)$ and the arguments in series around $\zeta = 1$, we can get a series expansion for $\sigma - \lambda$.

Here is the operator product of the argument of $\Psi(\zeta)$, $\mathrm{Ad}(\lambda) \to$ o[λ], $\mathrm{Ad}(\mu) \to$ o[μ].

```
o1 = OP[1 + o[λ] +  o[λ]^2/2,
        1 + t o[μ] + t^2 o[μ]^2/2] // Expand
```

This expression is the series expansion (we discuss series expansions in Chapter 1 of the Symbolics volume [140✱]) of $\Psi(z)$ around $z = 1$.

```
ser = Series[z Log[z]/(z - 1), {z, 1, 2}] // Normal
```

Now, we replace the individual terms in the series.

```
(a1 = ser /. {(-1 + z) -> o1 - 1, (-1 + z)^n_ :> 𝒪𝒫[(o1 - 1)^n]} //
                                             Expand) // Short[#, 10]&
```

Next, we carry out (by pattern matching) the `t`-integration.

```
a2 = a1 /. {t^n_. -> 1/(n + 1)};
```

We keep only terms up to order 3.

```
DeleteCases[
Which[FreeQ[#, 𝕠[_]], #,
      Length[Cases[{#}, f_. 𝕠[_]]] == 1, #,
      Length[Cases[{#}, f_. 𝒪𝒫[_]]] == 1,
       If[Total[(List @@ #[[2]]) /. {𝕠[_]^n_. -> n}] < 4,
          #, Null]]& /@ a2, (* these terms will be dropped *)
           _ Null | f_. 𝕠[μ] | f_. 𝒪𝒫[a___, 𝕠[μ]^n_.]]
```

Taking into account $[\mu, \mu] = 0$ and the definition above $\text{Ad}(\zeta)\,\eta = [\zeta, \eta]$, this gives:

$$\sigma - \lambda = \mu + \frac{1}{2}[\lambda, \mu] + \frac{1}{12}[\lambda, [\lambda, \mu]] - \frac{1}{12}[\mu, [\lambda, \mu]] + \cdots$$

For the convergence of this expansion, see [16✱]. For some related operator calculations in *Mathematica*, see [15✱]; for time-ordered generalizations, see [53✱]; for q-versions, see [129✱], [110✱].

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

b) We start by implementing the necessary operations between the noncommuting variables x and y. The noncommutative multiplication is this time denoted by o.

```
(* o goes through o *)
o[a___, o[xy__], b___] := o[a, xy, b];
(* factor out numerical factors *)
o[a___, f_ c_, b___] := Expand[f o[a, c, b]] /;
              FreeQ[f, x | y, {0, Infinity}, Heads -> True];
o[a___, f_, b___] := Expand[f o[a, b]] /;
              FreeQ[f, x | y, {0, Infinity}, Heads -> True];
(* pure powers are neighbors *)
o[a___, x^ex_., y^ey_., b___] := o[a, x^(ex - 1), x, y, y^(ey - 1), b];
(* powers of sums *)
o[a___, (p_Plus)^e_, b___] := o[a, p^(e - 1), p, b];
(* the fundamental commutation rule *)
o[a___, x, y, b___] := q o[a, y, x, b]
(* multiply out neighboring Plus terms *)
o[a___, p1_Plus, p2_Plus, b___] := o[a, Sum[o[p1[[i]], p2[[j]]],
                          {i, Length[p1]}, {j, Length[p2]}], b];
(* collect powers of y and powers of x *)
o[a___, x^e1_., x^e2_., x1___] := o[a, x^(e1  + e2), x1] /;
                         FreeQ[{x1}, y] && FreeQ[{a}, x];
o[y1___, y^e1_., y^e2_., b___] := o[y1, y^(e1 + e2), b] /;
                         FreeQ[{b}, y] && FreeQ[{y1}, x];
(* additivity *)
o[a___, p_Plus, b___] := Sum[o[a, p[[i]], b], {i, Length[p]}];
```

The function O helps to format the result nicely.

```
O[f_] :=
Module[{res = o[f]},
        (res //. (* or shorter:
      Collect[res, Cases[res, _o, Infinity], Factor] *)
            HoldPattern[a_. o[xy__] + b_. o[xy__]] :> (a + b) o[xy]) //.
                HoldPattern[a_ o[xy__]] :> Factor[a] o[xy]]
```

Here are two examples of the canonicalized form of the right-hand side of the q-binomial theorem.

```
O[(x + y)^3]
```

```
O[(x + y)^4]
```

We implement the right-hand side of the q-binomial theorem.

```
qBinomial[n_, k_, q_] :=
 qFactorial[n, q]/(qFactorial[k, q] qFactorial[n - k, q])

qFactorial[k_, q_] := Product[1 - q q^i, {i, 0, k - 1}]

qBinomialTheoremRhs[n_, q_] :=
  Sum[qBinomial[n, k, q] o[y^k, x^(n - k)], {k, 0, n}]
```

For $n = 1$, the theorem holds.

```
O[(x + y)^1] - qBinomialTheoremRhs[1, q]
```

For $n = 2$, we have to cancel common factors in rational functions in q.

```
O[(x + y)^2] - qBinomialTheoremRhs[2, q]
```

```
Simplify[%]
```

In a similar way, we can show the correctness of the theorem for higher n.

```
Table[Simplify[O[(x + y)^n] - qBinomialTheoremRhs[n, q]], {n, 1, 8}]
```

All rules for o are stored as down values for o in the form `HoldPattern[o[…] :> o[…]]`. We add a counting function `count` on the right-hand side of the `RuleDelayed`.

```
(* keep a copy of the original definitions *)
oldoDownValues = DownValues[o];
```

```
SetAttributes[count, HoldAll];
(* count increments the counter c by 1 *)
count[c_] := (c = c + 1);
```

The function `addCounter` actually splices the counter function `count` into the right-hand side of the `RuleDelayed`.

```
addCounter[rhs_ :> lhs_, k_, l_] := rhs :> (count[k]; lhs);

addCounter[rhs_ :> Verbatim[Condition][lhs_, cond_], k_, l_] :=
            rhs :> (Condition[count[k]; lhs, count[l]; cond])
```

Here are the new definitions counting how often the individual rules of o are applied. The counter for the kth rule is just `counter[k]`.

```
DownValues[o] =
 Table[addCounter[DownValues[o][[i]], counter[i], conditionCounter[i]],
      {i, Length[DownValues[o]]}]
```

We initialize all counters to 0.

```
Do[counter[i] = conditionCounter[i] = 0, {i, Length[DownValues[o]]}];
```

Now, we calculate `O[(x + y)^10]`.

```
O[(x + y)^10]
```

Here is the number of times the rules were applied.

```
??counter
```

`conditionCounter` shows the number of times the conditions were tested.

```
??conditionCounter
```

We restore the original definitions for *o*.

```
DownValues[o] = oldoDownValues;
```

Now let us deal with the two *q*-differentiation formulas. `qD` is the *q*-version of `D`.

We use the `qD[f_, n_, x_, q_] := qD[f, n, x, q]=...` construction to avoid the repeated evaluation of `qD[`*f(x)*`, `*n*`, `*x*`, `*q*`]`.

```
(* q-derivative; syntax similar to D *)
qD[f_, x_, q_] := (f - (f /. x -> q x))/((1 - q) x)

qD[f_, n_, x_, q_] := qD[f, n, x, q] = Nest[Factor[qD[#, x, q]]&, f, n]
```

The check of the two identities is straightforward for small *n*. We use `Factor` to show that all of the sums of nested fractions all.

```
Table[(* Together or *) Factor[qD[f[x] g[x], n, x, q] -
       Sum[qBinomial[n, k, q] (qD[f[x], n - k, x, q] /. x -> q^k x)*
           qD[g[x], k, x, q], {k, 0, n}]], {n, 0, 10}]

Table[(* Together or *) Factor[qD[f[x], n, x, q] - 1/(1 - q)^n 1/x^n*
      Sum[qBinomial[n, k, q] (-1)^k q^(-k(n - k) -k(k - 1)/2) f[x q^k],
          {k, 0, n}]], {n, 0, 10}]
```

For generalizations of the *q*-derivative, see [76✱], [55✱], [99✱], [102✱]. For differential operator representations of $\frac{d_q f(x)}{d_q x}$, see [101✱].

For dealing with the *h*-binomial theorem, we have to change just one definition of *o*, namely, the commutation relation.

```
o[a___, x, y, b___] := Expand[o[a, y, x, b] + o[a, h y^2, b]]

O[(x + y)^6]
```

Proceeding like above, it is straightforward to verify the first 10 instances of the *h*-binomial theorem.

```
(* the right-hand side of the h-binomial theorem *)
hBinomialTheoremRhs[n_, h_] :=
  Sum[Binomial[n, k] h^k Pochhammer[1/h, k] o[y^k, x^(n - k)],
      {k, 0, n}]

O[(x + y)^1] - hBinomialTheoremRhs[1, h] // Simplify

O[(x + y)^2] - hBinomialTheoremRhs[2, h] // Simplify

Table[Simplify[O[(x + y)^n] - hBinomialTheoremRhs[n, h]],
      {n, 2, 8}]
```

As a small side track, to make things more interesting for the not *q*-diseased readers, let us carry out an animation showing arguments of the entries of the *q*-Pascal triangle as *q* varies over the unit circle.

```
(* define q-Binomial *)
qBinomial[n_, k_, q_] :=
 qFactorial[n, q]/(qFactorial[k, q] qFactorial[n - k, q])

(qFactorial[k_, q_] := qFactorial[k, q] = #[k, q])&[
       Compile[{{k, _Integer}, {q, _Complex}},
               Product[1 - q^(i + 1), {i, 0, k - 1}]]]

(* q-Pascal triangle graphic *)
qBinomialArgPicture[nMax_, φq_, opts___] :=
Show[Graphics[(* color with phase *)
   Table[{Hue[(1 + Arg[qBinomial[n, k, Exp[1. I φq]]]/Pi)/2],
          Rectangle[{k - n/2, -n} - 1/2, {k - n/2, -n} + 1/2]},
        {n, 0, nMax}, {k, 0, n}]],
        opts, PlotRange -> All, Frame -> True,
        FrameTicks -> None, AspectRatio -> Automatic]

nMax = 120; frames = 17;
Show[GraphicsArray[qBinomialArgPicture[nMax, #,
                    DisplayFunction -> Identity]& /@ #]]& /@
Partition[Table[φq, {φq, 2Pi/frames, 2Pi(1 - 1/frames), 2Pi/frames}], 4]
```

```
nMax = 120; frames = 113;
Do[qBinomialArgPicture[nMax, φq],
   {φq, 2Pi/frames, 2Pi(1 - 1/frames), 2Pi/frames}];
```

For related generalizations of the multinomial coefficients, see [121✱] and [47✱].

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

c) The implementation of the “ordered derivative” is straightforward. $o[x_{\alpha_1},\ldots,x_{\alpha_n}]$ represents the operator product $\hat{x}_{\alpha_1}\,\hat{x}_{\alpha_2}\,\ldots\,\hat{x}_{\alpha_n}$ and $c[x_{\alpha_1},\ldots,x_{\alpha_n}]$ the string of classical symbols.

```
δ[o[l___], c[]] := Times[l]

δ[o[l_], c[x___]] := D[l, x]

δ[o[f_, g__], x_c] := With[{n = Length[x]},
          Sum[δ[o[f], x[[Table[j, {j, k}]]]] *
              δ[o[g], x[[Table[j, {j, k + 1, n}]]]], {k, 0, n}]]

δ[o[l___], c[x_]] := Product[D[{l}[[k]], x], {k, Length[{l}]}]
```

Here are some examples showing δ at work for two symbols x and p.

```
δ[o[x^2, p, x], c[x, p]]

δ[o[x^4, p, x^2, p, x, p], c[x, p]]
```

Now let us deal with the example given in the exercise text.

```
δ[o[x1 x2^2 x3^3 x4^4 x5^5 x6^4 x7^3 x8^2 x9],
  c[x1, x2, x3, x4, x5, x6, x7, x8, x9]]
```

`ReplaceList` conforms that there are exactly 2880 possibilities to delete the symbols $x_1x_2x_3x_4x_5x_6x_7x_8x_9$ from the string $x_1x_2x_2x_3x_3x_3x_4x_4x_4x_4x_5x_5x_5x_5x_5x_6x_6x_6x_6x_7x_7x_7x_8x_8x_9$.

```
ReplaceList[{x1, x2, x2, x3, x3, x3, x4, x4, x4, x4,
             x5, x5, x5, x5, x5, x6, x6, x6, x6,
             x7, x7, x7, x8, x8, x9},
  {___, x1, ___, x2, ___, x3, ___, x4, ___, x5,
   ___, x6, ___, x7, ___, x8, ___, x9, ___} :> 1] // Length
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

d) We model the noncommutative matrix multiplication with the head *o* and differentiation with respect to *t* with *d*. Both operations are linear and for the differentiation, we implement the Leibniz formula for products. We insert rule application counters on the right-hand side of each definition. We denote the inverse of a matrix **A** by *I*[**A**].

```
(* general properties of a noncommutative product o *)
o[a___, o[b___], c___] := (ro1 = ro1 + 1; o[a, b, c])
o[a___, f_?NumericQ o[b___], c___] := (ro2 = ro2 + 1; f o[a, b, c])
o[a___, b_ + c_, d___] := (ro3 = ro3 + 1; o[a, b, d] + o[a, c, d])

(* general properties of differentiation *)
d[o[a_, b__]] := (rd1 = rd1 + 1; o[d[a], b] + o[a, d[o[b]]])
d[o[a_]] := (rd2 = rd2 + 1; d[a])
d[a_ + b_] := (rd3 = rd3 + 1; d[a] + d[b])
d[f_?NumericQ a_] := (rd4 = rd4 + 1; f d[a])

(* differentiation of an inverse function *)
d[I[a_]] := (rd5 = rd5 + 1; -o[I[a], d[a], I[a]])
```

To count the rule applications we implement a counter initializing function `initializeCounters` and to view the number of rule applications, a function `ruleUsage`.

```
initializeCounters := (ro1 = ro2 = ro3 = rd1 = rd2 = rd3 = rd4 = rd5 = 0);
ruleUsage := {ro1, ro2, ro3, rd1, rd2, rd3, rd4, rd5}
```

We start by calculating $\partial^2 \mathbf{A}(t)^{(-1)} / \partial t^2$.

```
initializeCounters; d[d[I[A]]]
```

Here are the counts for the various rule applications.

```
ruleUsage
```

Next, we look at $\partial^3 \mathbf{A}(t)^{(-1)} / \partial t^3$. The result is getting larger.

```
initializeCounters; Nest[d, I[A], 3] // Expand

ruleUsage
```

For a nicer-looking result, we implement a function `shorten` that forms powers of matrices and unites derivatives. We also implement some typesetting rules for inverses and matrix products.

```
(* unite powers *)
shorten[expr_] := expr //.
   {o[a___, B:((b_)..), c___] :> o[a, b^Length[{B}], c] /;
         Length[{B}] > 1 && If[{a} =!= {}, Last[ {a}] =!= b, True] &&
                               If[{c} =!= {}, First[{c}] =!= b, True],
    d[d[a_]] :> Subscript[d, 2][a],
    d[Subscript[d, k_][a_]] :> Subscript[d, k + 1][a],
    Subscript[d, k_][Subscript[d, l_][a_]] :> Subscript[d, k + l][a],
    Subscript[d, k_][d[a_]] :> Subscript[d, k + 1][a]}
```

```
(* typeset definitions *)
With[{sf = StandardForm, sb = SuperscriptBox, s = Subscript},
MakeBoxes[o[args__], sf] := RowBox[{Sequence @@
       Drop[Flatten[Table[{MakeBoxes[#]&[{args}[[k]]],
                              "."}, {k, Length[{args}]}]], -1]}];
MakeBoxes[I[a_], sf] := sb[MakeBoxes[a],
                             RowBox[{"(", RowBox[{"-", "1"}], ")"}]];
MakeBoxes[d[a_], sf] := sb[MakeBoxes[a], "′"];
MakeBoxes[s[d, k_][a_], sf] := MakeBoxes[#]&[Derivative[k][a]]]
```

Now, we will calculate $\partial^5 \left(\mathbf{A}(t)^{(-1)}\right)^5 \big/ \partial t^5$.

```
initializeCounters;
D = Nest[d, o[I[A], I[A], I[A], I[A], I[A]], 5] // Expand;
```

The result has 681 terms and 110636 rule applications were carried out in the calculation.

```
{Length[D], ruleUsage}
```

Here are the first and last four terms of the 681 terms of the last result. (We could refine the function `shorten` to not only form powers of single matrices, but also of identical sequences of matrices.)

```
shorten @ D[[{1, 2, 3, 4}]]
```

```
shorten @ D[[{-1, -2, -3, -4}]]
```

Assuming commutativity means that the head o can be replaced with `Times`. Here is the commutative version of $\mathcal{D}$.

```
makeCommutativeRules =
  {o -> Times, Subscript[d, k_][a_] :> D[a[t], {t, k}],
   d[a_] :> D[a[t], t], I[a_]^n_. :> a[t]^-n};
```

```
shorten[D] //. makeCommutativeRules
```

Of course, it agrees with the direct derivative of $\partial^5 A(t)^{-5} \big/ \partial t^5$.

```
% - D[A[t]^-5, {t, 5}]
```

For closed forms for derivatives of matrix inverses, see [127✱].

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

9. Patterns and Replacements

a) We simply permute two arbitrary elements of the last constructed list, and append this new list to the lists in the already-constructed list of permutations, provided that it has not already been constructed, and has not remained unchanged during the exchange of the two elements. Because the tests are to be applied later, we should use `RuleDelayed`. The construction `{a___, b_, c___, d_, e___}` makes sure that all possible orders are taken into account.

```
allPermutations[li_List] :=
       {li} //. {{A___List, B:{a___, b_, c___, d_, e___}} :>
             ({A, B, {a, d, c, b, e}} /; (* avoid doubling *)
                  FreeQ[{A}, {a, d, c, b, e}] && B =!= {a, d, c, b, e})}
```

Here are a few examples.

```
allPermutations[{a, b, c}]
```

```
allPermutations[{1, 2, 3, 4}]
```

```
Length[%]
```

When we have repeated elements, fewer lists are generated.

```
allPermutations[{a, a, c}]
```

This implementation for generating permutations is, of course, not the most effective one; actually, *Mathematica* has the built-in command `Permutations`, which we will discuss in the next chapter (for algorithms to generate permutations, see [124✱]).

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

b) Starting with an initial list with only one number $\neq 0$ in the first place, we keep "pushing" a 1 to the right as long as the resulting list has not already been constructed, and as long as the list remains in descending order.

Again, we apply `RuleDelayed`.

```
allOrderedSplittings[li_List] :=
       {li} //. {{A___List, B:{a___, b_, c___, d_, e___}, C___List} :>
              ({A, B, {a, b - 1, c, d + 1, e}, C} /; b - 1 >= d + 1 &&
                     FreeQ[{A, C}, {a, b - 1, c, d + 1, e}] &&
                     OrderedQ[-{a, b - 1, c, d + 1, e}])}
```

Here again are two examples.

```
allOrderedSplittings[{5, 0, 0, 0, 0, 0}]

allOrderedSplittings[{13, 0, 0}]
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

c) We proceed in two steps. Starting with a list of the form {*oldList*, *stillEmptyList*, *newElements*}, we search for the first nonzero element, and replace it by the corresponding new element, whereas at the same time adding the same number of zeros in the new list to be constructed. The second replacement rule inside of the first group deals with the case in which no nonzero element is present. Then, the remaining (now empty) first and third lists are cut off.

```
replacementList[oldList_List, newElements_List] :=
({oldList, {}, newElements} //.
 {{{a___?(# == 0&), b_?(# != 0&), c___}, {d___}, {e_, f___}} ->
  {{c}, {d, a, e}, {f}},
  {{a___?(# == 0&)}, {d___}, {}} -> {{}, {d, a}, {}}}) /.
                                       (* remove by now empty working lists *)
                                              {{}, a_, {}} -> a
```

Here again are two examples.

```
replacementList[{0, 0, 0, 1, 0, 0, 0, 2, 0, 0, 0, 4},
                 {a,       b,           c,           d}]

replacementList[{1, 0, 0, 2, 0, 0, 3, 3, 0, 4, 0, 0, 5, 0, 0, -1, 0, 0},
                 {M,       A,       B, C,    D,       E,        N      }]
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

d) Again, we proceed in two steps. First, we generate a list containing the list of the starting numbers in the first place and the pairs of numbers with no common factors in the second place without taking into account their order (we identify these by the fact that the corresponding fraction cannot be reduced). In the second step, we remove the initial list.

```
pairGenerator[li_List?(VectorQ[#, Head[#] == Integer && # > 0 &]&)] :=
          (({li, {}} //.  (* first step *)
                   {{l:{a___, b_, c___, d_, e___}, {a___}} :>
                ({l, {a, {b, d}}} /; (* first condition *)
                   (FreeQ[{a}, {b, d}] &&
                    {b, d} == {Numerator[b/d], Denominator[b/d]})) }) //.
                        (* second step *)
                      {{l:{a___, b_, c___, d_, e___}, {a__}} :>
                 ({l, {a, {d, b}}} /; (* second condition *)
                   (FreeQ[{a}, {d, b}] &&
                    {b, d} == {Numerator[b/d], Denominator[b/d]})) }) /.
                         (* remove by now empty working list *)
                 {{_, l_} -> l}
```

Here are some relative prime pairs.

```
pairGenerator[{4, 2, 5, 3, 4, 4}]
```

In the following input, all numbers have the common divisor 2.

```
pairGenerator[{36, 30, 34, 18}]
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

e) The idea is to remove successive columns from the left, and measure the remaining number of rows to determine the length of each column. We implement the deletion of a column by subtracting 1 from each number. If a number reaches 0, it means the corresponding row is empty, and we delete it. Here, the first column is removed.

```
{{5, 3, 2, 1}} /. ({l___List, r_List} :> {l, r, r - 1 //.
                       {posInts___, Repeated[0]} -> {posInts}})
```

If we iterate this process, it ends naturally after n_1 steps.

```
{{5, 2, 1}} //. ({all___List, last:{__}} :> {all, last, last - 1 //.
                       {posInts___, Repeated[0]} -> {posInts}}) /.
                        {l__List, {}} -> {l}
```

Now, every sublist represents one constellation in the process of throwing away columns from the left, and the length of every sublist gives the length of the column.

```
% //. {{alreadyComputedLengths___Integer,
        subList_List, rest___List} :>
       {alreadyComputedLengths, Length[subList], rest}}
```

We put it all together and define.

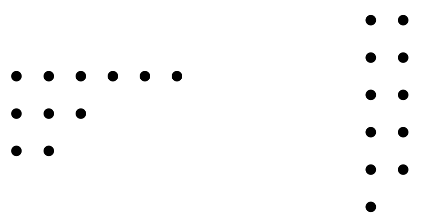
```
FerrerConjugate[li:{_Integer..}] :=
(({li} //. ({all___List, last:{__}} :> {all, last, last - 1 //.
    {posInts___, Repeated[0]} -> {posInts}}) /.
         {l__List, {}} -> {l}) //.
           {{alreadyComputedLengths___Integer,
             subList_List, rest___List} :>
             {alreadyComputedLengths, Length[subList], rest}}) /;
                                                 OrderedQ[-nuli]
```

Here are two examples.

```
FerrerConjugate[{6, 3, 2}]
```

```
FerrerConjugate[{2, 2, 2, 2, 2, 1}]
```

The last two results are easily verified with the following pictures.



To generate the last two arrays of points programmatically, we could use the following.

```
Show[GraphicsArray[#, GraphicsSpacing -> -0.2]]& @
(Graphics[{{AbsolutePointSize[8], Point[#]}& /@ Flatten[
 MapIndexed[Transpose[{Range[#], Table[-#2[[1]], {#1}]}]&, #], 1]},
           AspectRatio -> Automatic, PlotRange -> All]& /@
           {{6, 3, 2}, {2, 2, 2, 2, 2, 1}})
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

10. Hermite Polynomials, Peakons

a) Here is a possible implementation. Note the use of `TagSet` (because of the product structure on the left-hand side), and the application of `Expand`. Both are needed to apply the rule a multiple number of times.

```
H /: x_^m_Integer?Positive H[n_, x_] :=
                  Expand[(n H[n - 1, x] + 1/2 H[n + 1, x])x^(m - 1)]

H /: x_ H[n_, x_] := (n H[n - 1, x] + 1/2 H[n + 1, x])
```

Here is the result of the program.

```
Table[Expand[x^n H[m, x]], {n, 0, 4}]
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

b) It is straightforward to define of the Camasso–Holm differential operator. But the solution does not give immediately the result 0.

```
CamassaHolmOperator[ψ_, {x_, t_}] := D[ψ, t] - D[ψ, x, x, t] +
      3 ψ D[ψ, x] - 2 D[ψ, x] D[ψ, x, x] - ψ D[ψ, x, x, x]

ψ[x_, t_] := c Exp[-Abs[x - c t]]

CamassaHolmOperator[ψ[x, t], {x, t}] // Simplify
```

The last result contains unevaluated derivatives of the function `Abs`. While the absolute value function is differentiable along the real axis, it is not differentiable as a function of a complex variable, *Mathematica*'s default domain. If instead of `Abs`, we use the on-the-real-axis-equivalent function $(x^2)^{1/2}$, we get the expected result.

```
abs[x_] = Sqrt[x^2];

ψ[x_, t_] := c Exp[-abs[x - c t]]

CamassaHolmOperator[ψ[x, t], {x, t}] // Simplify
```

The function $\psi(x, t)$ is not differentiable at $x = c\,t$ where it has a cusp (the solution is a so-called peakon [24✱], [86✱], [87✱], [64✱], [48✱], [85✱], [108✱], [90✱]). There the derivative of the function `abs` is undefined.

```
D[abs[x], x]
```

We remedy this shortcoming by using a differentiable approximation $|x|_\alpha$ of $|x|$, such that $\lim_{\alpha\to\infty} |x|_\alpha = |x|$, and show that for all α the function $\psi(x, t)$ fulfills the differential equation at $x = c\,t$.

```
abs[x_, α_] = -x + Log[1 + Exp[2 α x]]/α;
```

```
ψ[x_, t_] := c Exp[-abs[x - c t, α]]
```

```
CamassaHolmOperator[ψ[x, t], {x, t}] /. x -> c t
```

Here is system of two partial differential equations in two spatial and one temporal variables.

```
BroerKaupEquations =
{D[H[x, y, t], t, y] == D[H[x, y, t], x, x, y] -
                        2 D[H[x, y, t] D[H[x, y, t], x], y] -
                        2 D[G[x, y, t], x, x],
 D[G[x, y, t], t] == -D[G[x, y, t], x, x] -
                      2 D[G[x, y, t] H[x, y, t], x]};
```

And here is a solution that contains arbitrary functions $p(x, t)$ and $q(y)$ that allow to build 2D peakons [7✱].

```
HSol[x_, y_, t_] := (c1 + C q[y]) D[p[x, t], x]/
                    (1 + c1 p[x, t] + c2 q[y] + C p[x, t] q[y]) -
                    (D[p[x, t], t] + D[p[x, t], x, x])/(2 D[p[x, t], x]);
```

```
GSol[x_, y_, t_] := (C - c1 c2) D[p[x, t], x] D[q[y], y]/
                 (1 + c1 p[x, t] + c2 q[y] + C p[x, t] q[y])^2
```

We quickly check that these two functions are solutions.

```
Simplify[BroerKaupEquations /. {H -> HSol, G -> GSol}]
```

For discrete peakons, see [32✱].

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

11. f[x___] := …

a) Here is our first function definition.

```
f[x___] := x + 1
```

Here, the argument is `Sequence[]`, so that the right-hand side of the function definition gives `Plus[1] = 1`.

```
f[]
```

Here the argument is `Sequence[1, 2, 3]`, so that the right-hand side of the function definition gives `Plus[1, 2, 3, 1] = 7`.

```
f[1, 2, 3]
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

b) Here is our second function definition.

```
f[x___] := x - 1
```

This expression is the internal form of the function definition.

```
DownValues[f] // FullForm
```

It differs from the first function definition only in that the last 1 is replaced by -1.

```
f[]
```

`f[1, 2, 3]` evaluates to `Plus[1, 3, 2, -1]`.

```
f[1, 2, 3]
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

c) The function definition is different in the third example. The `Subtract[x, 1]` is not rewritten as `Plus[x, -1]`.

```
f[x___] := Subtract[x, 1]

DownValues[f] // FullForm
```

Because `Subtract` needs exactly two arguments, we get an error message.

```
f[]

f[1, 2, 3]
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

d) Let us analyze the pattern on the left-hand side. The outer two occurrences of `HoldPattern` have no influence on later pattern matchings. They just avoid any evaluation of the pattern (in this case nothing would have been nontrivially evaluated anyway). The `HoldPattern` inside the `Verbatim` is of relevance. The `Verbatim` makes the left-hand side a definition for `HoldPattern[f]`. So the result of evaluating `HoldPattern[f]` is `4`.

```
f /: HoldPattern[HoldPattern[Verbatim[HoldPattern[f]]]] := 4
HoldPattern[f]
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

12. Result and Error Messages

We examine what happens in the evaluation, and interpret the generated error messages and the result.

```
{1, 2} //. {{x___, y___} :> ({x, Unique[c], y} /;
                             (Head[{x}[[-1]]] =!= Symbol &&
                              Head[{y}[[ 1]]] =!= Symbol))}
```

We begin with the result. According to the replacement rule, a `c$`*n* is to be inserted between any two non-`Symbols`, that is, in this case, between all numbers. To understand the origin of the error messages, we look at the interpretation `x` and `y` selected by *Mathematica* each time a replacement is attempted.

```
{1, 2} //. {{x___, y___} :> ({x, Unique[c], y} /;
                             (Print["x = ", x, "  and y = ", y];
                              Head[{x}[[-1]]] =!= Symbol &&
                              Head[{y}[[ 1]]] =!= Symbol))}
```

We can now see the problem. Because of the `BlankNullSequence` in the pattern, an interpretation of `x` as `Sequence[]` is possible. Using this result as an argument in `Sequence[][[-1]]` or `Sequence[][[1]]` leads to the following error.

```
{Sequence[]}[[-1]]

{Sequence[]}[[ 1]]
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

13. Patterns

a) This method is probably the most common way to define such a pattern, by the use of `PatternTest`.

```
f0[i_Integer?(2 <= # <= 8&)] := s[i];
```

```
{f0[1], f0[2], f0[3], f0[4], f0[5], f0[6], f0[7], f0[8], f0[9]}
```

We can also give a `Condition`. In the following definition of `f1`, the second argument of `SetDelayed` has the form `Condition[`*expr*`, `*test*`]`.

```
f1[i_Integer] := s[i] /; 2 <= i <= 8;
```

```
{f1[1], f1[2], f1[3], f1[4], f1[5], f1[6], f1[7], f1[8], f1[9]}
```

But the `Condition` can also appear in the first argument of `SetDelayed`.

```
f2[i_Integer /; 2 <= i <= 8] := s[i];
```

```
{f2[1], f2[2], f2[3], f2[4], f2[5], f2[6], f2[7], f2[8], f2[9]}
```

Next, we could think of various mixtures of `Condition` and `PatternTest`, like in this example.

```
(f3[i_Integer?(# >= 2&)] /; i <= 8) := s[i];
```

```
{f3[1], f3[2], f3[3], f3[4], f3[5], f3[6], f3[7], f3[8], f3[9]}
```

In the case of interest here, the number of all possible arguments could also be given explicitly in an `Alternatives`-construction.

```
f4[i:(2 | 3 | 4 | 5 | 6 | 7 | 8)] := s[i];
```

```
{f4[1], f4[2], f4[3], f4[4], f4[5], f4[6], f4[7], f4[8], f4[9]}
```

A variety of further possibilities exist, like this one, in which a second pattern only matches in the case when it is absent; that is, the length of all of its pieces is 0.

```
f5[i:(2 | 3)  | i_Integer?(# >= 4&), j___?(Length[{#}] === 0&)] :=
                                                       s[i] /; i < 9
```

```
{f5[1], f5[2], f5[3], f5[4], f5[5], f5[6], f5[7], f5[8], f5[9]}
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

b) This is the function `f`.

```
f[Condition[Condition_, Condition; True],
  Optional[Blank_, Optional],
  Pattern[Pattern, Blank[Integer]],
  Four:(4 | 4.),
  PatternTest[Pattern[PatternTest, Blank[]], PatternTest; True&],
  Alternatives:Alternatives[Alternatives, 6],
  Flat_Flat,
  Stub:Blank[Orderless[OneIdentity]],
  HoldPattern_HoldPattern?(# === #&),
  HoldPattern[Set[3, 4]]] :=
      {Condition, Blank, Pattern, Four, PatternTest,
       Alternatives, Flat[[1]], Stub[[1]], 9, HoldPattern[[1]], 11}
```

The first argument has the form `Condition[Condition_, Condition; True]`. It is a `Condition` that is always fulfilled, and the pattern variable is again `Condition`. So the first argument has to be `1`.

```
f1[Condition[Condition_, Condition; True]] := Condition
```

```
f1[1]
```

The third argument is `Optional[Blank_, Optional]`. This argument is optional. To get the value 2 for it, we should have for the pattern variable `Blank` the value `2`.

```
f2[Optional[Blank_, Optional]] := Blank
```

```
f2[2]
```

The second pattern is of the form `Pattern[Pattern, Blank[Integer]]`. Here, `Pattern` has to be an integer; this is the case for `3`.

```
f3[Pattern[Pattern, Blank[Integer]]] := Pattern
```

```
f3[3]
```

The fourth variable has to be `4` or `4.0`. We use the `4`.

```
f4[Four:(4 | 4.)] := Four
```

```
f4[4]
```

The fifth argument is represented by the pattern `PatternTest[Pattern[PatternTest, Blank[]], Pat⋱ ternTest; True&]`. Again, this argument is always `True` giving `PatternTest`. The pattern variable is this time `PatternTest`, and we can use just `5` as the fifth argument.

```
f5[PatternTest[Pattern[PatternTest, Blank[]], PatternTest; True&]] := Patte
```

```
f5[5]
```

The sixth pattern is `Alternatives:Alternatives[Alternatives, 6]`. Now, `Alternatives` is the pattern variable used for either `Alternatives` or `6`. We use the `6`.

```
f6[Alternatives:Alternatives[Alternatives, 6]] := Alternatives
```

```
f6[6]
```

The seventh pattern is `Flat_Flat`, which means the argument has to have the head `Flat` to match the pattern. To simultaneously get our `7`, we use `Flat[7]` as the argument, because fortunately the right-hand side of the definition for `f` specifies that the first element has to be taken.

```
f7[Flat_Flat] := Flat[[1]]
```

```
f7[Flat[7]]
```

The eighth pattern is `Stub:Blank[Orderless[OneIdentity]]`, which means that the head of the argument must have the compound head `Orderless[OneIdentity]`. Again, the first part is extracted on the right-hand side, and we use `Orderless[OneIdentity][8]` as the eighth argument.

```
f8[Stub:Blank[Orderless[OneIdentity]]] := Stub[[1]]
```

```
f8[Orderless[OneIdentity][8]]
```

The ninth argument in the definition of `f` is `HoldPattern_HoldPattern?(# === #&)`. Because the `Pattern⋱ Test` always yields `True` for this tautological test, we have to use an argument in which the head is `HoldPattern` and whose first argument is `1`.

```
f9[HoldPattern_HoldPattern?(# === #&)] := HoldPattern[[1]]
```

```
f9[HoldPattern[10]]
```

The last argument must match the pattern `HoldPattern[Set[3, 4]]`. Because `f` has no attribute like `Hold`, we must avoid the evaluation of the argument, which can be achieved with `Unevaluated`.

```
f10[HoldPattern[Set[3, 4]]] := matches

f10[Unevaluated[Set[3, 4]]]
```

So, we finally have this result.

```
f[1, 2, 3, 4, 5, 6, Flat[7], Orderless[OneIdentity][8],
  HoldPattern[10], Unevaluated[Set[3, 4]]]
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

c) In the first example, the `a` from `Pattern[a, Blank[]]` on the left-hand side of the definition of `a` is the local variable, which is fed into `Unique` when calling `a[`*argument*`]`. Then, a new variable is created, which is used as the variable in `Function[#, Hold[#], HoldAll]`. This pure function, having the attribute `HoldAll`, returns the whole left-hand side (the pattern `f`) enclosed in `Hold`. For the inputs `a[a]` and `a[b]`, this works fine, but in case of the arguments `2a` or `a + a`, the argument does not have the head `Symbol` or `String`, but instead `Times` or `Plus`. So `Unique` cannot create a new variable, an error message is created, and the construction `Unique[2a]` (in the case of `a + a`, the addition is carried out inside `Unique`, because at this time no attributes prevent the evaluation) cannot be used as a variable inside `Function`, so that the result is `Function[Unique[2a], Hold[Unique[2a]], HoldAll][a[2 a]]`. Here, we see the calculation carried out.

```
SetAttributes[a, HoldAll]

f:a[a_] := Function[#, Hold[#], {HoldAll}][f]&[Unique[a]]

{a[a], a[b], a[2a], a[a + a]}
```

In the second example, the unique variable created by `Unique` is created completely independent of the argument of the left-hand side, because now the argument of `Unique` is a string. So the calculation can be done for all four arguments. Also, the last case remains completely unevaluated.

```
Remove[a]

SetAttributes[a, HoldAll]

f:a[a_] := Function[#, Hold[#], {HoldAll}][f]&[Unique["a"]]

{a[a], a[b], a[2a], a[a + a]}
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

d) Here the calculation is carried out.

```
SetAttributes[AtomQ, HoldAll]

{AtomQ[1/2], AtomQ[1 + I]}
```

Because of the `HoldAll` attribute, the arguments are not evaluated before they are passed to `AtomQ`. But in an unevaluated form, `1/2` is not `Rational[1,2]` but rather `Times[1, Power[2, -1]]`, which is not an atom. Similarly, the unevaluated form of `1 + I` is not in `Complex[1, 1]`, but `Plus[1, I]`, which again is not an atom.

```
FullForm[Hold[1/2]]

FullForm[Hold[1 + I]]
```

Using `Unevaluated`, we can directly pass the arguments to `AtomQ`, without giving `AtomQ` explicitly the attribute `HoldAll`.

```
ClearAttributes[AtomQ, HoldAll]

{AtomQ[Unevaluated[1/2]], AtomQ[Unevaluated[1 + I]]}
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

e) Let us run the input under consideration.

```
blank[Pattern[Blank, Blank[Blank]]] = Blank;

blank[Blank[Blank]]
```

We use `blank[Pattern[Blank, Blank[Blank]]] = Blank` to make a definition for the function `blank`. The first argument in `Pattern` is the name of the local pattern variable; here it is `Blank`. The second argument of `Pattern` is the pattern-object. Any actual argument of `blank` given later must match this pattern-object. In the case under consideration, it is `Blank[Blank]` (or shorter `_Blank`); this is a pattern standing for any expression (the outer `Blank` in `Blank[Blank]` with the head `Blank` (the inner `Blank` in `Blank[Blank]`). The argument `Blank[Blank]` in `blank[Blank[Blank]]` matches the pattern in the definition (it has head `Blank`). The definition for `blank` defines the result of `blank[`*x*`]` in case *x* has head `Blank` to be just *x*; here this is `Blank[Blank]`. This is the result we obtained above in its output form `_Blank`.

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

f) The first definition works as expected. For an argument less than -1, `f1` prints `C1` and `C2` and returns `Null`. For an argument greater than -1, the function `f1` prints just `C1` and returns the input.

```
f1[x0_] := Block[{x = x0}, Print[C1]; x = x + 1; Print[C2] /; Positive[x]]
f1[-2]
```

The definition of `f1` shows nothing unexpected.

```
DownValues[f1] // FullForm
```

But how can the definition act this way? How does *Mathematica* know that a construction of the form *f*`[`*x*`_] := Block[{`*localVars*`}, `*body* `/; `*condition*`]` means a condition of the applicability of *f* rather than returning an expression with head `Condition`? We see the magic behind this evaluation by using `Trace`.

```
Trace[f1[-2]]
```

The expression that was evaluated was not
`f1[x0_] := Block[{x = x0}, Print[C1]; x = x + 1; Print[C2] /; Positive[x]]`
but rather
`Block[{x = -2},Print[C1]; x = x + 1; RuleCondition[Print[C2], Positive[x]]]`.

When a `Condition` in the last argument of the `CompoundExpression` that forms `Block`'s body is explicitly present *Mathematica* introduces, from the beginning of the evaluation, a new function, namely `RuleCondition`. `RuleCondition` gets always formed when a condition is explicitly present at the end of `Block`, `Module`, or `With`. Because it is typically not explicitly input, it is considered to be an internal symbol.

```
??RuleCondition
```

In the definition of `f2`, the function `Condition` is not explicitly present in the body of the `Block`. Only at runtime, it gets created. But at this time, no `RuleCondition` statement can be created anymore. Because of the `HoldAll` attribute of `Condition` and the nonuse of `Condition` in a definition here, the value of `x` gets not used in `Positive[x]` and `Null /; Positive[x]` is returned. Because there are no restrictions in the evaluation of the body of the `Block`, `C1`, and `C2` are printed.

```
f2[x0_] := Block[{x = x0}, ToExpression[
                 "Print[C1]; x = x + 1; Print[C2] /; Positive[x]"]]
f2[-2]
```

In the definition of `f3`, the function `Condition` is present in the body of `Block`, but not in such a way that a `Rule`· `Condition` is formed. `Condition` is present as a symbol, not as a function with arguments. Evaluating the body starts with evaluating the compound expression. Its result is `condition[Null, False]`. As side effects, the variables `C1` and `C2` are printed out. Then the replacement `condition -> Condition` is carried out and `Null /; False` is the result.

```
f3[x0_] := Block[{x = x0}, (Print[C1]; x = x + 1;
                            condition[Print[C2], Positive[x]]) /.
                                       condition -> Condition]
f3[-2]
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

14. Replacements

a) The list `{1, 2, 3, 4, 5}` matches the pattern. The condition `b > 2` is not satisfied for the pattern realization `b = 2`, so the result of the application of the `RuleDelayed` is again `{1, 2, 3, 4, 5}`. Therefore, no change occurred and `ReplaceRepeated` ends the substitution.

```
{1, 2, 3, 4, 5} //. {a__, b_, c_, d___} :>
                      If[b > 2, {b, c, d}, {a, b, c, d}]
```

Using a `Print` statement on the right-hand side of the `RuleDelayed`, the matching pattern can be seen.

```
{1, 2, 3, 4, 5} //. {a__, b_, c_, d___} :>
                      (Print[{{a}, {b}, {c}, {d}}];
                       If[b > 2, {b, c, d}, {a, b, c, d}])
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

b) In this case, the pattern matches again (although with another realization). The condition `b > 2` is again not satisfied, and the result of the application of the rule is the original expression.

```
{1, 2, 3, 4, 5} //. {a___, b_, c_, d___} :>
                  If[b > 2, {b, c, d}, {a, b, c, d}]
```

Again, using `Print`, we see the pattern realizations tried.

```
{1, 2, 3, 4, 5} //. {a___, b_, c_, d___} :>
                  (Print[{{a}, {b}, {c}, {d}}];
                        If[b > 2, {b, c, d}, {a, b, c, d}])
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

c) Now, the condition `b > 2` is implemented via `Condition`. The first matching pattern found (`a = {1, 2}, b = {3}, c = {4}, d = {5}`) is applied. Using `ReplaceAll` we see this first result.

```
{1, 2, 3, 4, 5} /. {a__, b_, c_, d___} :> {b, c, d} /; b > 2
```

Then, the rule is applied again (with the matching `a = {3}, b = {4}, c = {5}, d={}`). The result is again the list `{4, 5}`, which does not match the pattern `{a__, b_, c_, d___}`, so the application of the rule stops here.

```
{1, 2, 3, 4, 5} //. {a__, b_, c_, d___} :> {b, c, d} /; b > 2
```

Using `Print` again, we see all tried patterns.

```
{1, 2, 3, 4, 5} //. {a__, b_, c_, d___} :>
             {b, c, d} /; (Print[{{a}, {b}, {c}, {d}}]; b > 2)
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

d) Again, the condition `b > 2` is implemented via `Condition`. The first matching pattern found (`a = {1, 2}, b = {3}, c = {4}, d = {5}`) is applied. Using `ReplaceAll`, we see this result.

```
{1, 2, 3, 4, 5} /. {a___, b_, c_, d___} :> {b, c, d} /; b > 2
```

Then, the rule is applied again (with the matching `a = {}, b = {3}, c = {4}, d = {5}`). The result is again the list `{3, 4, 5}`, so the application of the rule stops here.

```
{1, 2, 3, 4, 5} //. {a___, b_, c_, d___} :> {b, c, d} /; b > 2
```

Again, using `Print`, we see all matching trials.

```
{1, 2, 3, 4, 5} //. {a___, b_, c_, d___} :>
                     {b, c, d} /; (Print[{{a}, {b}, {c}, {d}}]; b > 2)
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

e) In this example, two conditions are present. Let us first look at the structure of the rule itself.

```
FullForm[Hold[thePattern /; cond1 :> res /; cond2]]
```

```
{1, 2, 3, 4, 5} //. (({a__, b_, c_, d___} /; b > 2) :> {b, c, d} /; b > 2)
```

The condition on the left-hand side does not add a new condition, so this example is equivalent to the one from part c) and the result is again the list `{4, 5}`.

To see all intermediate steps, we use now two `Print` statements, one on the left-hand side of the rule and one on the right-hand side of the rule.

```
{1, 2, 3, 4, 5} //. (({a__, b_, c_, d___} /;
   (Print[{lhs, {{a}, {b}, {c}, {d}}}]; b > 2)) :>
          {b, c, d} /; (Print[{rhs, {{a}, {b}, {c}, {d}}}]; b > 2))
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

f) Again, the condition on the left-hand side does not add a new condition, so this example is equivalent to the one from part d) and the result is again the list `{3, 4, 5}`.

```
{1, 2, 3, 4, 5} //. (({a___, b_, c_, d___} /; b > 2) :> {b, c, d} /; b > 2)
```

We again use two `Print` statements, one on the left-hand side of the rule and one on the right-hand side of the rule.

```
{1, 2, 3, 4, 5} //. (({a___, b_, c_, d___} /;
    (Print[{lhs, {{a}, {b}, {c}, {d}}}]; b > 2)) :>
        {b, c, d} /; (Print[{rhs, {{a}, {b}, {c}, {d}}}]; b > 2))
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

15. Puzzles

a) Here are two possible solutions. The first possibility is to give `a` the property that a call to `a` changes its truth value from `True` to `False`.

```
(aWasCalled = False; a := (aWasCalled = Not[aWasCalled]))
```

```
a
```

```
And[a, a]
```

A second possibility is to add a special rule to `And`. (Because `True` has the attribute `Locked`, we cannot give an upvalue for `True`.)

```
Remove[a]; $Line = 0;

(Unprotect[And]; And[True, True] = False; a = True;)

a

And[True, True]
```

Another possibility would be to use an upvalue for `a`. Because `And` is `HoldAll`, this is easily possible.

```
(a /: And[a, a] = False); a = True;

a

And[a, a]
```

In the next possible `And[a, a]`-fake, we manipulate the result with `$Post`.

```
Remove[a]; $Line = 0;

$Post = If[$Line > 2, False, True]&;

a

And[a, a]
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

b) Here, the input is carried out. We restart *Mathematica* here.

```
(Im[3 I] =!= 3) // Function[{x}, Block[{I}, x], {HoldAll}]
```

We got an error message. The error was generated because a locked symbol cannot be localized with `Block`.

```
Block[{I}, x]

Block[{Symbol}, x]
```

But `I` was considered as a symbol; before its evaluation, it is the symbol `I`, and after its evaluation, it is the complex number `Complex[0, 1]`.

```
Hold[I] // FullForm

I // FullForm
```

The localization would have worked inside a `Module` or a `With`.

```
Module[{I}, Im[3 I] =!= 3]

With[{I = 1}, Im[3 I] =!= 3]
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

c) We evaluate the input under consideration.

```
Hold[With[{z = Abort[]}, z^2]] /. z_?Quit :> Quit[]
```

Let us discuss in detail what is happening. The head of the whole expression is `ReplaceAll`.

```
Hold[Hold[With[{z = Abort[]}, z^2]] /. z_?Quit :> Quit[]] // FullForm
```

`ReplaceAll` evaluates its first and second argument. The first one is a `Hold`, and the second one is a `RuleDe-layed`, so nothing dangerous happens.

```
Hold[With[{z = Abort[]}, z^2]]

z_?Quit :> Quit[]
```

Now, the replacement happens. Here, we look at the elements matched to `z_`.

```
Remove[z];

Hold[With[{z = abort[]}, z^2]] /. z_?(Print[{#, z}]&) :> Quit[]
```

```
Abort[]^2
```

The third call is the one causing the abort to happen.

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

d) Here is the complete calculation monitored with `On[]`.

```
On[]; 2/3 === Unevaluated[2/3]
```

```
Off[]
```

The seeming paradox that things look the same, but are not, is easy to explain: Using `On[]`, all intermediate steps are given in `OutputForm`; `Rational[2, 3]` (the result of the left-hand side) and `Times[2, Power[3, -1]]` look the same in an ordinary call to `OutputForm`.

```
Unevaluated[2/3] // OutputForm
```

```
FullForm[%]
```

```
Rational[2, 3] // OutputForm
```

Using `Trace` and looking at the result in `FullForm` also shows that `SameQ` gets `Rational[2, 3]` and `Times[2, Power[3, -1]]` as arguments.

```
FullForm /@ Trace[2/3 === Unevaluated[2/3] ]
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

e) The following simple program searches for all such symbols.

```
allBuiltInSymbols = Names["*"];

Do[temp = ToExpression[allBuiltInSymbols[[i]]];
   If[Not[TrueQ[temp == temp]], Print[allBuiltInSymbols[[i]]]],
   {i, Length[allBuiltInSymbols]}]
```

Only two symbols have this property, namely, `Indeterminate` and `ComplexInfinity`.

```
{Indeterminate == Indeterminate,
 ComplexInfinity == ComplexInfinity}
```

The reason for this behavior is to avoid a misleading `True` for equations (head `Equal`) of the form a `==` b, where both a and b evaluate to `Indeterminate` or `ComplexInfinity`.

```
{(1 - 1)/(2 - 2) == (1 - 1)^(2 - 2), 1/0 == I/0}
```

`SameQ` gives `True`. It tests if two expressions are equal as *Mathematica* expressions, whereas `Equal` cares about mathematical equality.

```
{Indeterminate === Indeterminate,
 ComplexInfinity === ComplexInfinity}
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

f) The result will be `𝕐`.

```
(𝕏[_?(# === _?#0&), C_ /; MatchQ[C, _ /; MatchQ[C, _]]] := 𝕐;
 𝕏[_?(# === _?#0&), C_ /; MatchQ[C, _ /; MatchQ[C, _]]])
```

The two patterns in the definition both have the property that they match themselves. The first one represents a pattern in which the pattern test must reproduce the whole pattern by using #0.

```
MatchQ[_?(# === _?#0&), _?(# === _?#0&)]
```

The second argument in the definition has the condition on the pattern that it is itself a condition.

```
MatchQ[C_ /; MatchQ[C, _ /; MatchQ[C, _]],
       C_ /; MatchQ[C, _ /; MatchQ[C, _]]]
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

g) IntegerQ returns True or False when it is called with one argument. With zero, or two, or more arguments it stays unevaluated. *x* can evaluate to zero, or two, or more arguments when it has head Sequence.

```
x := Sequence[];
IntegerQ[x]
```

```
x := Sequence[1, 2, 3];
IntegerQ[x]
```

Another possibility for IntegerQ[*x*] is having *x* be a compound expression that sets up or modifies existing definitions. For instance, we could make a new upvalue for, say, j and then call IntegerQ with the argument j.

```
x := (j /: f_[j] := f; j);
IntegerQ[x]
```

Or we could actually manipulate the definition of IntegerQ itself inside the argument of IntegerQ. Because IntegerQ does not have a Hold-like attribute, its argument gets evaluated and the new rule goes into effect before the outer IntegerQ evaluates with its argument.

```
x := (Unprotect[IntegerQ]; IntegerQ[_] := IntegerQ);
IntegerQ[x]
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

h) Here is the mentioned iteration limit problem shown.

```
$IterationLimit = 20;
```

```
SetAttributes[f, {Flat, OneIdentity}]
f[b_] := b
f[a, b]
```

The problem is caused by f[a, b] matching the pattern of the definition f[ξ_] := ξ because of the Flat and OneIdentity attribute. ξ evaluates to itself and this leads to the iteration problem. The following input demonstrates this by keeping the argument unevaluated inside the function g (we give g the HoldAll attribute).

```
Remove[f, a, b]
SetAttributes[f, {Flat, OneIdentity}]
SetAttributes[g, HoldAll]
f[b_] := g[b]
f[a, b]
```

To avoid the iteration we must restrict the application of the definition to the case where f is called with genuinely one argument. This can be done by using either Condition or PatternTest or inside Block. In addition to make f[ξ] evaluate to ξ we have to extract the ξ carefully from the unevaluated one-argument form of f when not using Block. Here are three possibilities shown. All three make the one-argument form of f work, avoid the iteration problem in the two-argument version, and at the same time keep all the properties related to the Flat attribute alive.

```
Remove[f, a, b]
SetAttributes[f, {Flat, OneIdentity}]
F:_f := Block[{f}, First[F] /; Length[F] === 1]

{f[a], f[a, b], f[f[a], f[b, c]]}

Remove[f, a, b]
SetAttributes[f, {Flat, OneIdentity}]
F:_f := Hold[F][[1, 1]] /; Length[Unevaluated[F]] === 1

{f[a], f[a, b], f[f[a], f[b, c]]}

Remove[f, a, b]
SetAttributes[f, {Flat, OneIdentity}]
F:_f?(Function[f, Length[Unevaluated[f]] === 1, {HoldAll}]) :=
                                              Unevaluated[F][[1]]

{f[a], f[a, b], f[f[a], f[b, c]]}
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

i) We start with the definition `f[HoldPattern[HoldPattern][x_]] = x`. To match this pattern the innermost `HoldPattern` must be present.

```
f1[HoldPattern[HoldPattern][x_]] = x;
{f1[HoldPattern[1]], f1[Verbatim[1]], f[1]}
```

Now let us consider the definition `f[HoldPattern[Verbatim][x_]] = x`. To match this pattern `Verbatim` must be present. The additional `HoldPattern` around the `Verbatim` in the function definition has no influence.

```
f2[HoldPattern[Verbatim][x_]] = x;
{f2[HoldPattern[1]], f2[Verbatim[1]], f[1]}
```

The third definition is `f[Verbatim[HoldPattern][x_]] = x`. `Verbatim[HoldPattern]` means that `HoldPattern` must occur verbatim in the argument.

```
f3[Verbatim[HoldPattern][x_]] = x;
{f3[HoldPattern[1]], f3[Verbatim[1]], f[1]}
```

The last definition is f`[Verbatim[Verbatim][x_]] = x`. `Verbatim[Verbatim]` means that `Verbatim` must occur verbatim in the argument.

```
f4[Verbatim[Verbatim][x_]] = x;
{f4[HoldPattern[1]], f4[Verbatim[1]], f[1]}
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

j) For the definition of `f` to go into effect, the first argument can be arbitrary and the second must be an assignment with `Set`. The left-hand side of this assignment must have head `g` and the argument of `g` must coincide with the first argument of `f`. The right-hand side of the assignment for `g` must be `y^2` verbatim.

```
With[{a = x}, HoldPattern[f[y_, g[y_] = y^2]] := a]

??f

??g
```

To have the `Set` in the second argument we must use `Unevaluated`.

```
f[z, Unevaluated[g[z] = y^2]]
```

The assignment for `g` was never evaluated.

This means any expression that evaluates to itself, not just a symbol, can be used for `y$`.

```
f[a nonsymbol, Unevaluated[g[a nonsymbol] = y^2]]
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

k) The next input evaluates the expression under consideration.

```
Block[{Function}, (#&[2]) /. Function -> Print]
```

`Block` has the local variable `Function`. This means that the typical properties of `Function` will not be active inside `Block`. So `#&[2]` does not evaluate to `2`, but rather stays unchanged. Then the replacement `Function -> Print` is carried out. The argument of `Function` was `#1`, and so `Slot[1]` is printed. The result of the evaluated `Print` statement is `Null` and the `Block` statement returns `Null[2]`. Using a local variable other than `Function`, the pure function in the body evaluates to `2` and `Function` is no longer present anymore to be replaced.

```
Block[{function}, (#&[2]) /. function -> Print]
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

l) Here is what happens when evaluating the inputs.

First, two definitions are set up for the one- and two-argument form of `M`. Then `M[x]` is evaluated. This generates an upvalue for `x`. This upvalue matches any expression of the form `e:h_[___,x, ___]`, meaning any expression containing `x` at level 1. (If such an expression is found, the right-hand side of the upvalue evaluates. When doing this, *e* is printed and the original upvalue definition for `x` is destroyed, `M[h, e]` is evaluated and *e* is returned.) Then `α[1, β[y], a[b[c[d[f[x]]]]]]` is evaluated. The three arguments of α are evaluated in order and when evaluating the third argument the subexpression `f[x]` is found. This causes the upvalue for `x` to go into effect, and as a result `M[f, x]`, will be evaluated. The two-argument form of M works similarly to the one-argument form. `M[h, e]` creates an upvalue definition for `e:ℓ_[___, e, ___]`. (When the right-hand side of this definition is evaluated, *e* is printed and the original upvalue definition for *h* is destroyed, `M[ℓ, e]` is evaluated and *e* is returned.) So after evaluating `f[x]` an upvalue for `f` of the form `f /: e:ℓ_[___, f[x], ___]` is in effect. When `d[f[x], 1]` is evaluated this upvalue definition fires, `d[f[x], 1]` is printed, and a new upvalue definition for `d` is generated. This process continues with the heads `c`, `b`, `a`, and finally α.

Here we carry out the inputs under consideration.

```
SetAttributes[{M, TagUnset, ToString}, HoldAllComplete]

M[e_] := (e /: HoldPattern[e:h_[___, e, ___]] :=
          (Print["Found: ", h, " ", HoldForm[e]];
           ToExpression[# <> " /: HoldPattern[e:h_[___, " <>
                             # <> ", ___]] =."]&[ToString[e]];
           M[h, e]; e))

M[h_, e_] := (h /: HoldPattern[e:ℓ_[___, e, ___]] :=
               (Print["Found: ", HoldForm[e]];
                TagUnset @@ {h, UpValues[h][[1, 1, 1]]}; M[ℓ, e]; e))

M[x];
α[1, β[y], a[b[c[2, d[f[x], 1]]]]]

ClearAttributes[{TagUnset, ToString}, HoldAllComplete]
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

m) The function does indeed implement the condition for separability in a straightforward way. And whenever `separableVariablesQ` will return `True` for a function *f*, it will be surely separable. Here is an example.

```
separableVariablesQ[f_, {x_, y_}, {x0_, y0_}] :=
 (Simplify[f /. {x -> x0, y -> y0}] =!= 0) &&
  Simplify[f D[f, x, y] - D[f, x] D[f, y]] === 0
```

```
separableVariablesQ[(Cos[x - y] - Cos[x + y])/2, {x, y}, {1, 2}]
```

The problem with the function `separableVariablesQ` is when it returns `False`. As a function ending in `Q`, it must (for the correct number of arguments) return `True` of `False`. The construct `And[UnsameQ[…], SameQ[…]]` guarantee this. But it might happen that `Simplify` does not succeed showing that `f D[f, x, y] - D[f, x] D[f, y]` is zero. And indeed, we can always make a function structurally inseparable by a term of the form $x + zero\, y$. If zero is a sufficiently complicated zero (and some theorems guarantee that we can always find such zeros), then `Simplify` cannot resolve this zero and we will get the answer false from `False` from `separableVariablesQ`, although the function was separable. Here is an example of this situation.

```
zero = Sqrt[2 + Sqrt[2 + Sqrt[2]]]/2 - Cos[Pi/16];
```

```
separableVariablesQ[(Cos[x - y] - Cos[x + y (1 + zero x)])/2,
                    {x, y}, {1, 2}]
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

n) We start by observing that 2 + 3 *i* is a Gaussian prime.

```
PrimeQ[2 + 3 I, GaussianIntegers -> True]
```

And that `PrimeQ` has, by default, the attribute `Listable`.

```
Attributes[PrimeQ]
```

To predict the result of the input under consideration, we must remember the evaluation order discussed in the last chapter. After the evaluation of the `SetAttributes`-input, the function `PrimeQ` has the attribute `HoldAll`. This means its two arguments `2 + 3 I` and are not immediately `{GaussianIntegers -> True}` evaluated. The `Listable` attribute results in `{PrimeQ[2 + 3 I, GaussianIntegers -> True]}`. Now `PrimeQ` goes to work. Its first argument is still `Plus[2,Times[3,I]]`, meaning an expression with head `Plus`, not a number. Because being a number is mandatory for being a prime number, the `PrimeQ[...]` evaluates to `False` and the result returned is `{False}`.

```
SetAttributes[PrimeQ, HoldAll]
PrimeQ[2 + 3 I, {GaussianIntegers -> True}]
```

If we force the evaluation of the first argument of `PrimeQ`, we obtain the result `{True}`.

```
PrimeQ[Evaluate[2 + 3 I], {GaussianIntegers -> True}]
```

Without the `HoldAll` attribute, but again with an unevaluated argument, we get again the result `{False}`.

```
ClearAttributes[PrimeQ, {HoldAll}];
PrimeQ[Unevaluated[2 + 3 I], {GaussianIntegers -> True}]
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

o) The five elements of the output characterize the result as “unusual”.

The result of `In[2]` shows that `In[1]` was a relatively short numeric expression (the fourth test basically measures the length of the input in characters) that was not a number but contained inexact numbers. The shortness of the input and the absence of user symbols from the `Global`` context indicate that the input could not contain any `Set`-`Attributes`- or `TagSet`-operation to associate an artificial property with a user symbol. (Also, faking a built-in symbol using, say, `Symbol`a` gives already a too long input.) In addition, the context analysis of the input shows only built-in symbols. So the input must have been a short input using a built-in function (there is hardly room for using two functions) with the NumericFunction attribute, that, for approximate numbers does not evaluate to a number. While there are built-in functions with the NumericFunction attribute, that do not evaluate to numbers for all arguments because of restricted domains of definitions (such as UnitStep[I] or DedekindEta[-I]), and they all have longer names than needed here. The shortest built-in functions with the NumericFunction attribute are the two-letter functions Re and Im. When the argument is a single approximate number, they surely evaluate to a number. But for two arguments, no built-in rules exist and the expression stays unevaluated. But the NumericFunction attribute still makes them a numeric expression (in the sense of NumericQ). And indeed, the following input has all the properties we were looking for. (When evaluated, we get an additional message because *Mathematica* does not expect Re to be called with two arguments.)

```
Re[1., 1]
```

```
{NumericQ[%], NumberQ[%], MemberQ[%, _?InexactNumberQ],
 StringLength[StringDrop[ToString[
                         DownValues[In][[$Line - 1]]], 22]],
 Context /@ Cases[%, _Symbol, {-1}, Heads -> True]}
```

There are plenty of modifications of this input that yield identical results for the In[2] from above.

```
Im[2., E]
```

Σ (* session summary *) **TMGBs`PrintSessionSummary[]**

p) If a subexpression b explicitly literal in the tree form of a, then Position will surely find its position (assuming the option setting for the Heads option is identical for FreeQ and Position). So, we must rely on the nonliteral presence of b in a. In this case, it is obviously impossible for Position to return a result other than {}. Because FreeQ allows patterns as its second argument and takes attributes of functions into account, we can construct the following example where b is not literally present, but is present after taking the attributes into account.

```
SetAttributes[f, {Flat, Orderless}];
a = f[x, y, z]; b = f[x, z];
{FreeQ[a, b], Position[a, b]}
```

Σ (* session summary *) **TMGBs`PrintSessionSummary[]**

16. Evaluation Sequence

In the first definition, the Condition does not matter at all for the function definition because it is not part of a definition, but rather wrapped around a complete definition.

```
(f[x_] := g) /; c
```

```
?f
```

```
Clear[f, c]
```

```
(f[x_] := g) /; (Print[c]; c)
```

In the second definition, we have the condition on the left-hand side of the definition.

```
Clear[f]
```

```
(f[x_] /; c) := g
```

```
?f
```

We can see the order of evaluation by adding additional Print statements.

```
Clear[f, c]

(f[x_?((Print[#]; True)&)] /; (Print[c]; c = True)) := (Print[g]; g)

f[x]
```

We see that first the pattern, then the condition, and then the right-hand side become evaluated.

Using `Print` statements again, we see that the same evaluation sequence happens for the third and fourth definitions.

```
Clear[f];

(f[x_] := g /; c)

?f

Clear[f, c]

f[x_?((Print[#]; True)&)] := (Print[g]; g) /; (Print[c]; c = True)

f[x]

Clear[f];

f[x_ /; c] := g

?f

Clear[f, c]

f[(x_?((Print[#]; True)&)) /; (Print[c]; c = True)] := (Print[g]; g)

f[x]
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

17. Nested Scoping

a) Applying the function `f` to the argument `y` just replaces all instances of `x` in the definition of `f` by `y`.

```
Clear[f]; f[x_] := Function[x, x]; f[y]
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

b) `With` replaces every nonscoped instance of the local variables in the body by the corresponding value, which means the two `x`s in the `Function` will be replaced by `z`.

```
With[{x = z}, Function[x, x]]
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

c) The replacement rule again replaces the two “`x`”s in the `Function` with `z`.

```
Function[x, x] /. x -> z
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

d) The function definition with `SetDelayed` inside the `Function` keeps the `x` local, and the resulting definition contains `x`, not `y`.

```
Function[x, f[x_] := x^2][y]; DownValues[f]
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

e) `With` does not replace scoped variables, which means the two “`x`”s in the function definition will be not replaced by y.

```
With[{x = y}, f[x_] := x^2]; DownValues[f]
```

Σ (* session summary *) **TMGBs`PrintSessionSummary[]**

f) A literal replacement of y by the outer x would mostly not do what we want, so the scoped x in the inner functions gets renamed to x$.

```
Function[y, Function[x, x + y]][x]
```

Σ (* session summary *) **TMGBs`PrintSessionSummary[]**

g) The same renaming happens in the following application of f.

```
f[y_] := Function[x, x + y]; f[x]
```

Σ (* session summary *) **TMGBs`PrintSessionSummary[]**

h) The x and the z in the left-hand side of the SetDelayed definition are not pattern variables, so they just get their local values inside the Module. The x on the left-hand side of the SetDelayed is local to Function and gets renamed.

```
Module[{x, y, z = a}, f[x, y_, z] := Function[x, x + y + z]];
DownValues[f]
```

Σ (* session summary *) **TMGBs`PrintSessionSummary[]**

i) In this last example, first the z gets substituted everywhere. The rest is similar to the last example.

```
With[{z = a}, Module[{x, y}, f[x, y_, z] := Function[x, x + y + z]]];
DownValues[f]
```

Σ (* session summary *) **TMGBs`PrintSessionSummary[]**

18. Why {b,b}?

After the Table has been evaluated, the Union goes to work. The list to Union has only as or only bs or both. Assume as and bs occur. Then, Union unions them to {a, b}. After Union has finished its job, the Date[] might have advanced and the a in {a, b} is now evaluated to b.

```
a := b /; EvenQ[Last[Date[]]]
```

We carry out the Table command 20 times; sometimes the result is {b} and sometimes {b, b}.

```
Table[Union[Table[a, {10000}]], {20}]
```

Σ (* session summary *) **TMGBs`PrintSessionSummary[]**

References

✱1 R. M. Abu-Sarris in S. Elaydi, F. Allan, A. Elkhader, T. Mughrabi, M. Saleh (eds.). *Proceedings of the Mathemat-ics Conference*, World Scientific, Singapore, 2000. *BookLink*

✱2 M. S. Alber, C. Miller. *arXiv:nlin.PS*/0001004 (2000). *Get Preprint*

✱3 G. E. Andrews. *SIAM Rev.* 16, 441 (1974).

✱4 G. E. Andrews. *The Theory of Partitions*, Addison-Wesley, Reading, 1976. *BookLink (3)*

✱5 L. Anné, P. Joly, Q. H. Tran. *Comput. Geosc.* 4, 207 (2000). *DOI-Link*

✱6 R. Askey. *CRM Proc. Lecture Notes* 9, 13 (1996).

✱7 C. Bai, H. Zhao. *Chaos, Solitons, Fractals* 23, 777 (2004). *DOI-Link*

✱8 A. D. Bandrauk, H. Shen. *J. Chem. Phys.* 99, 1185 (1993). *DOI-Link*

✱9 A. Banerjee. *arXiv:quant-ph*/0502163 (2005). *Get Preprint*

✱10 M. P. Barnett. *ACM SIGSAM Bull.* 37, 49 (2003). *DOI-Link*

✱11 K. J. Baumeister in S. Sengupta, J. Häuser, P. R. Eiseman, J. F. Thompson (eds.). *Numerical Grid Generation in Computational Fluid Mechanics*, Pineridge Press, Swansea, 1988. *BookLink*

✱12 H. B. Benaoum. *J. Phys.* A 31, L 751 (1998). *DOI-Link*

✱13 H. B. Benaoum. *arXiv:math-ph*/9812028 (1998). *Get Preprint*

✱14 H. B. Benaoum. *J. Phys.* A 32, 2037 (1999). *DOI-Link*

✱15 V. N. Beskrovnyi. *Comput. Phys. Commun.* 111, 76 (1998). *DOI-Link*

✱16 S. Blanes, F. Casas. *Lin. Alg. Appl.* 378, 135 (2004). *DOI-Link*

✱17 D. Bonatsos, C. Daskaloyannis. *arXiv:nucl-th*/9999003 (1999). *Get Preprint*

✱18 J. M. Borwein, D. M. Bradley, R. E. Crandall. *J. Comput. Appl. Math.* 121, 247 (2000). *DOI-Link*

✱19 A. Bose. *J. Math. Phys.* 30, 2035 (1989). *DOI-Link*

✱20 D. Bowman. *q-Difference Operators, Orthogonal Polynomials, and Symmetric Expansions*, American Mathematical Society, Providence, 2002. *BookLink*

✱21 B. Buchberger in P. Gaffney, E. N. Houstis, A. Hilt (eds.). *Programming Environments for High-Level Scientific Problem Solving*, North Holland, Amsterdam, 1992. *BookLink*

✱22 B. Buchberger. *An Implementation of Gröbner Bases in Mathematica*, *MathSource* 0205-300 (1992).

✱23 C. S. Calude, M. J. Dinneen, C.-K. Shu. *arXiv:nlin.CD*/0112022 (2001). *Get Preprint*

✱24 R. Camassa, D. D. Holm. *Phys. Rev. Lett.* 71, 1661 (1993). *DOI-Link*

✱25 R. Camassa, D. L. Holm, J. M. Hyman. *Adv. Appl. Mech.* 31, 1 (1994).

✱26 S. Chandrasekhar. *Rev. Mod. Phys.* 15, 1 (1943). *DOI-Link*

✱27 K. Charter, T. Rogers. *Exper. Math.* 2, 209 (1994).

✱28 S. Ciliberti, G. Caldarelli, P. D. L. Rios, L. Pietronero, Y.-C. Zhang. *Phys. Rev. Lett.* 85, 4848 (2000). *DOI-*

Link

✶29 H. Cirstea, C. Kirchner. *INRIA Report* RR-3818 (1999). http://www.inria.fr/RRRT/RR-3818.html

✶30 M. W. Coffey. *J. Comput. Appl. Math.* 166, 525 (2004). *DOI-Link*

✶31 H. Cohn. *Am. Math. Monthly* 111, 487 (2004).

✶32 A. Comech, J. Cuevas, P. G. Kevrekidis. *arXiv:nlin.PS*/05020002 (2005). *Get Preprint*

✶33 L. Comtet. *Advanced Combinatorics*, Reidel, Dordrecht, 1974. *BookLink (2)*

✶34 B. Costa, W. S. Don. *Appl. Num. Math.* 33, 151 (2000). *DOI-Link*

✶35 J. Czyz. *J. Geom. Phys.* 6, 595 (1989).

✶36 A. Degasperis, D. D. Holm, A. N. W. Hone. *arXiv:nlin.SI*/0205023 (2002). *Get Preprint*

✶37 A. Degasperis, D. D. Holm, A. N. W. Hone. *arXiv:nlin.SI*/0209008 (2002). *Get Preprint*

✶38 H. De Raedt. *Comput. Phys. Rep.* 7, 1 (1987). *DOI-Link*

✶39 J. Derbyshire.*Prime Obsession*, Joseph Henry Press, Washington, 2003. *BookLink (2)*

✶40 A. Dimakis, F. Müller–Hoissen. *Phys. Lett.* B 295, 242 (1992). *DOI-Link*

✶41 L. Di Vizio. *arXiv:math.NT*/0211217 (2002). *Get Preprint*

✶42 V. K. Dobrev. *arXiv:quant-ph*/0207077 (2002). *Get Preprint*

✶43 H. M. Edwards. *Riemann*'*s Zeta Function*, Academic Press, Boston, 1974. *BookLink (2)*

✶44 P. Erdös. *Discr. Math.* 136, 53 (1994). *DOI-Link*

✶45 J. Esch, T. D. Rogers. *Discr. Comput. Geom.* 25, 477, (2001).

✶46 H. Exton. *q-Hypergeometric Functions and Applications*, Ellis Horwood, Chichester, 1983. *BookLink*

✶47 B. L. Feigin, S. A. Loktev, I. Y. Tipunin. *Commun. Math. Phys.* 229, 271 (2002). *DOI-Link*

✶48 A. S. Fokas, P. J. Olver, P. Rosenau in A. S. Fokas and I. M. Gel'fand (eds.). *Progress in Nonlinear Differential Equations*, Birkhäuser, Boston, 1996. *BookLink*

✶49 B. Fornberg in G. D. Byrne, W. E. Schiesser (eds.). *Recent Developments in Numerical Methods and Software for ODEs/DAEs/PDEs*,World Scientific, Singapore, 1992. *BookLink*

✶50 B. Fornberg. *A Practical Guide to Pseudospectral Methods*, Cambridge University Press, Cambridge, 1996. *BookLink (2)*

✶51 B. Fornberg. *SIAM Rev.* 40, 685 (1998). *DOI-Link*

✶52 B. Fornberg, M. Ghrist. *SIAM J. Num. Anal.* 37, 105 (1999). *DOI-Link*

✱53 J. D. Franson, M. M. Donegan. *arXiv:quant-ph*/0108018 (2001). *Get Preprint*

✱54 L. Galue. *Algebras, Groups Geometries* 14, 83 (1997).

✱55 T. Golinski, A. Odzijewicz. *Czech. J. Phys.* 52, 1219 (2002). *DOI-Link*

✱56 I. P. Goulden, D. M. Jackson. *Combinatorial Enumeration*, Wiley, New York, 1983. *BookLink (2)*

✱57 A. Z. Górski, J. Szmigielski. *hep-th*/970315 (1997). *Get Preprint*

✱58 A. Z. Górski. *Acta Phys. Polonica* B 31, 789 (2000).

✱59 B. Green, T. Tao. *arXiv:math.NT*/0404188 (2004). *Get Preprint*

✱60 R. Grimshaw, B. A. Malomed, G. A. Gottwald. *arXiv:nlin.PS*/0203056 (2002). *Get Preprint*

✱61 M. M. Gupta, J. Kouatchou. *SIAM Rev.* 44, 83 (1998).

✱62 S. Hauswirth. *arXiv:hep-lat*/0003007 (2000). *Get Preprint*

✱63 A. S. Hegazi, M. Mansour. *Int. J. Theor. Phys.* 41, 1815 (2002). *DOI-Link*

✱64 D. D. Holm, M. F. Staley. *arXiv:nlin.CD*/0203007 (2002). *Get Preprint*

✱65 Q. Hou, N. Goldenfeld, A. McKane. *arXiv:cond-mat*/0009449 (2000). *Get Preprint*

✱66 A. Ivic. *The Riemann Zeta-Function*, Wiley, New York, 1985. *BookLink (2)*

✱67 D. Jacobson. *The Mathematica Journal* 2, n4, 42, (1992).

✱68 R. Jaganathan. *arXiv:math-ph*/0003018 (2000). *Get Preprint*

✱69 W. P. Johnson. *Discr. Math.* 157, 207 (1996). *DOI-Link*

✱70 A. A. Karatsuba. *Complex Analysis in Number Theory*, CRC Press, Boca Raton, 1995. *BookLink*

✱71 V. Kathotia. *Int. J. Math.* 11, 523 (2000).

✱72 E. Katz, U.-J. Wiese. *Phys. Rev.* E 58, 5796 (1998). *DOI-Link*

✱73 I. R. Khan, R. Ohba. *J. Comput. Appl. Math.* 107, 179 (1999). *DOI-Link*

✱74 T. H. Kjeldsen. *Arch. Hist. Exact Sci.* 56, 469 (2002). *DOI-Link*

✱75 S. Klarsfeld, J. A. Oteo. *J. Phys.* A 22, 4565 (1989). *DOI-Link*

✱76 M. Klimek. *J. Phys.* A 26, 955 (1993). *DOI-Link*

✱77 T. H. Koornwinder. *arXiv:math.CA*/9403216 (1994). *Get Preprint*

✱78 T. Koornwinder. *Informal Paper*(1999).
http://www.wins.uva.nl/pub/mathematics/reports/Analysis/koornwinder/qbinomial.ps

✱79 B. A. Kupershmidt. *J. Nonlin. Math. Phys.* 7, 244 (2000).

✱80 S. T. Kuroda in W. F. Ames, E. M. Harrell II, J. V. Herod (eds.). *Differential Equations with Applications to Mathematical Physics*, Academic Press, Boston, 1993. *BookLink*

✱81 C. S. Lam. *arXiv:hep-th*/9804181 (1998). *Get Preprint*

✱82 D. Larsson, S. D. Silvestrov. *J. Nonlin. Math. Phys.* 10, 95 (2003).

✱83 S. Levy. *The Mathematica Journal* 1, n3, 63, (1991).

✱84 X.-J. Li. *J. Number Th.* 65, 325 (1997). *DOI-Link*

✱85 Y. A. Li, P. J. Olver, P. Rosenau in M. Grosser, G. Hormann, M. Kunzinger, and M. Oberguggenberger (eds.). *Nonlinear Theory of Generalized Functions*, Chapman and Hall, New York, 1999. *BookLink*

✱86 Z. Liu, T. Qian. *Int. J. Bifurc. Chaos* 11, 781 (2001). *DOI-Link*

✱87 Z. Liu, T. Qian. *Appl. Math. Model.* 26, 473 (2002). *DOI-Link*

✱88 Z. Liu, R. Wang, Z. Jing. *Chaos, Solitons, Fractals* 19, 77 (2003). *DOI-Link*

✱89 M. Lothaire. *Algebraic Combinatorics on Words*, Cambridge University Press, Cambridge, 2002. *BookLink*

✱90 H. Lundmark, J. Szmigielski. *arXiv:nlin.SI*/0503036 (2005). *Get Preprint*

✱91 K. Maslanka. *arXiv:math.NT*/0402168 (2004). *Get Preprint*

✱92 K. Maurin. *The Riemann Legacy*, Kluwer, Dordrecht, 1997. *BookLink*

✱93 K. Mayrhofer, F. D. Fischer. *ZAMM* 74, 265 (1994).

✱94 S. A. Messaoudi. *Int. J. Math. Edu. Sci. Technol.* 33, 425 (2002). *DOI-Link*

✱95 G. A. Miller. *Am. Math. Monthly* 28, 116 (1921).

✱96 W. Miller Jr. *Symmetry Groups and Their Applications*, Academic Press, New York, 1972. *BookLink (2)*

✱97 J. Morales, A. Flores–Riveros. *J. Math. Phys.* 30, 393 (1989). *DOI-Link*

✱98 F. Neuman. *Adv. Difference Eq.* 2, 111 (2004).

✱99 A. Odzijewicz, T. Golinski. *arXiv:math-ph*/0208006 (2002). *Get Preprint*

✱100 J. A. Oteo. *J. Math. Phys.* 32, 419 (1991). *DOI-Link*

✱101 H. Pan, Z. S. Zhao. *Phys. Lett.* A 282, 251 (2001). *DOI-Link*

✱102 D. Parashar, D. Parashar. *J. Geom. Phys.* 48, 297 (2003). *DOI-Link*

✱103 T. Petersen. *The Mathematica Journal* 2, n4, 10, (1992).

✱104 P. A. Pritchard, A. Moran, A. Thyssen. *Math. Comput.* 64, 1337 (1995).

✱105 L. D. Pustyl'nikov. *Russ. Math. Surv.* 54, 262 (1999). *DOI-Link*

✱106 L. D. Pustyl'nikov. *Russ. Math. Surv.* 55, 207 (2000). *DOI-Link*

✱107 L. D. Pustyl'nikov. *Izvest. Math.* 65, 85 (2001).

✱108 Z. Qiao, X. B. Qiao. *Chaos, Solitons, Fractals* 25, 177 (2005). *DOI-Link*

✱109 R. Qu. *Math. Comput. Model.* 24, 55 (1996).

✱110 C. Quesne. *arXiv:math-ph*/0310038 (2003). *Get Preprint*

✱111 R. Reigada, A. H. Romero, A. Sarmiento, K. Lindenberg. *arXiv:cond-mat*/9905003 (1999). *Get Preprint*

✱112 M. W. Reinsch. *arXiv:math-ph*/9905012 (1999). *Get Preprint*

✱113 M. W. Reinsch. *J. Math. Phys.* 41, 2434 (2000). *DOI-Link*

✱114 P. Ribenboim. *Nieuw Archief Wiskunde* 12, 53 (1994).

✱115 R. D. Richtmeyer. *Principles of Advanced Mathematical Physics*, Springer-Verlag, Berlin, 1981. *BookLink*

✱116 R. D. Richtmeyer, S. Greenspan. *Commun. Pure Appl. Math.* 18, 107 (1965).

✱117 A. Riddle. *The Mathematica Journal* 1, n3, 60 (1991).

✱118 A. V. Ryzhov, L. G. Yaffe. *arXiv:hep-ph*/0006333 (2000). *Get Preprint*

✱119 J. M. Sanz-Serna, M. P. Calvo. *Numerical Hamiltonian Problems*, Chapman & Hall, London, 1994. *BookLink*

✱120 H. Scheffé. *Technometrics* 12, 388 (1970).

✱121 A. Schilling. *arXiv:q-alg*/9701007 (1997). *Get Preprint*

✱122 A. Schilling, S. O.Warnaar. *Ramanujan J.* 2, 459 (1998). *DOI-Link*

✱123 D. Scott. *Am. Math. Monthly* 92, 422 (1985).

✱124 R. Sedgewick. *Comput. Surveys* 9, 137 (1977).

✱125 C. Shu. *Differential Quadrature and its Applications in Engineering Sciences*, Springer-Verlag, Berlin, 2000. *BookLink*

✱126 J. Si-cong. *Chin. Sci. Bull.* 34, 1248 (1989).

✱127 M. Singh. *J. Phys.* A 23, 2307 (1990). *DOI-Link*

✱128 A. T. Sornborger, E. D. Stewart. *arXiv:quant-ph*/9903055 (1999). *Get Preprint*

✱129 R. Sridhar, R. Jagannathan. *arXiv:math-ph*/0212068 (2002). *Get Preprint*

✱130 R. P. Stanley. *Enumerative Combinatorics* v.1, Cambridge University Press, Cambridge, 1997. *BookLink (3)*

✱131 D. Stanton in E. Koelink, W. Van Assche (eds.). *Orthogonal Polynomials and Special Functions*, Springer-Verlag, Berlin, 2003. *BookLink*

✱132 S. Steinberg. *J. Diff. Eq.* 26, 404 (1977).

✱133 S. Steinberg, P. J. Roache in D. V. Shirkov, V. A. Rostovtsev, V. P. Gerdt (eds.). *IV. International Conference on Computer Algebra in Physical Research*, World Scientific, Singapore, 1991.

✱134 B. Strand. *J. Comput. Phys.* 110, 47 (1994). *DOI-Link*

✱135 H. Suyari. *arXiv:cond-mat*/0401546 (2004). *Get Preprint*

✱136 M. Suzuki. *Commun. Math. Phys.* 57, 193 (1977).

✱137 M. Suzuki. *Int. J. Mod. Phys.* C 7, 355 (1996). *DOI-Link*

✱138 E. C. Titchmarsh. *The Theory of the Riemann Zeta Function*, Clarendon Press, Oxford, 1986. *BookLink*

✱139 M. Trott. *The Mathematica GuideBook for Numerics*, Springer-Verlag, New York, 2005. *BookLink*

✱140 M. Trott. *The Mathematica GuideBook for Symbolics*, Springer-Verlag, New York, 2005. *BookLink*

✱141 D. R. Truax. *Phys. Rev.* D 31, 1988 (1985). *DOI-Link*

✱142 J. H. van Lint, R. M. Wilson. *A Course in Combinatorics*, Cambridge University Press, Cambridge, 1992. *BookLink (4)*

✱143 I. Vardi. *The Mathematica Journal* 1, n3, 63 (1991).

✱144 M. Veltman. *Nucl. Phys.* B 319, 253 (1989).

✱145 C. P. Viazminsky. *arXiv:math.NA*/0210167 (2002). *Get Preprint*

✱146 G. Walz. *Asymptotics and Extrapolation*, Akademie Verlag, Berlin, 1996. *BookLink (2)*

✱147 S. Weintraub. *J. Recreat. Math.* 18, 281 (1986).

✱148 B. D. Welfert. *SIAM J. Num. Anal.* 34, 1640 (1997). *DOI-Link*

✱149 S. Wolfram. *Mathematica: A System for Doing Mathematics by Computer*, Addison-Wesley, Reading, 1992. *BookLink*

✱150 K. Zarankiewicz. *Matematyka* 2, n4, 1 (1949).

✱151 K. Zarankiewicz. *Matematyka* 2, n5, 1 (1949).

P R O G R A M M I N G

CHAPTER **6**

Operations on Lists, and Linear Algebra

6.0 Remarks

This chapter on lists is the last chapter on the structure of *Mathematica* expressions and programming in *Mathematica*. We start presenting somewhat larger programs, especially in Sections 6.3.4, 6.4.4, 6.5.2, and 6.6. These programs deal mostly with mathematical, physical, and scientific/engineering applications of *Mathematica*, although some of them serve primarily to illustrate *Mathematica* as a programming language. At the outset, we do not place too much value on elegance, and we intentionally present classical procedural program segments. As we get deeper into the material, we will also make use of more elegant functional programming techniques. However, functional programming should not be overdone. From the standpoint of readability (for an example, see Subsection 2.3.10 of the Graphics volume [301✱]), it is sometimes better to introduce auxiliary variables, even when they make the program longer and are not needed. In addition, functional programs are often relatively complicated for the newcomer, although they can be much faster than a corresponding using procedural routine.

To save time and space and to improve readability, we will not always conduct the most desirable tests needed to determine if the variables passed to a procedure are appropriate. This testing can be done using *_head*, `PatternTest`, and `Condition`. Leaving out such tests has one advantage in the framework of the *GuideBooks*: It is frequently very instructive to call a given program segment with "inappropriate" arguments, say, symbolical instead of numerical and to study what happens in such situations. Moreover, we do not protect all programs and program segments from other programs in the chapter as well as we could have (using the constructions `Block`, `Module`, and `With` discussed in Chapter 4).

Usually, we restrict ourselves to generic cases. We do not try to make most programs work with a wider set of problems. Various special cases would have to be programmed to avoid, such as division by zero. Numeric and symbolic arguments would have to be treated separately for speed reasons, and so on.

The lists to be discussed in this chapter are very important objects in *Mathematica*. They represent sets, vectors, matrices, tensors, etc. Almost all larger data sets (they arise, for example, in images, in finding roots of larger polynomials, in solving equations, etc.) are collected in lists. Lists are "containers" for (potentially very large) data sets. Lists can be nested in a completely arbitrary way, independent of their size, depth, and content. *Mathematica* implements a large set of effective commands for manipulating lists. For nested lists (tensors) of machine integers, real numbers and complex numbers, *Mathematica* carries out appropriate optimizations by generating packed arrays (see Chapter 1 of the Numerics volume [302✱] of the *GuideBooks* for details). These commands include sorting, reordering, combining, and splitting lists, as well as various set theory operations. Because the basic objects of linear algebra, vectors, and matrices are also represented in *Mathematica* as lists, we discuss various mathematical operations, such as matrix multiplication, solution of systems of linear equations, eigenvalues.

List operations are useful and fast in *Mathematica* when dealing with large amounts of data. A typical example is the generation of a graphics image. Here is a routine `GluedPolygons` that recursively glues regular polygons to each other with a given angle between these normals (the argument `form` determines if the resulting faces should be rendered as holed polygons or as lines along the boundaries) and displays the resulting polygons (for details of the 3D graphics generation, see Chapter 2 of the Graphics volume [301✱]).

```
In[45]:= (* no spelling warnings, set fonts for tick labels, ... *)
         Get[ToFileName[ReplacePart["FileName" /.
          NotebookInformation[EvaluationNotebook[]], "Initialization.m", 2]]];

         GluedPolygons[n_Integer?(# >= 3&), angle:α_?(Im[N[#]] === 0&),
                       iter__Integer?(# >= 0&), faceShape:(Polygon | Line),
                       opts___Rule] :=
         Module[{c = N[Cos[α]], s = N[Sin[α]], myUnion, r, ℛ, allm, argch,
                makeHole, makeLine, n = #/Sqrt[#.#]&, ε = 10^-6},
         (* a completely transitive Union  *)
         myUnion[l_] := Union[l, SameTest -> ((Plus @@ (#.#& /@ (#1 - #2))) < ε&)];
         (* construction of next layer *)
         (* rotate a point *)
         r[point_, rotPoint_, {dir1_, dir2_, dir3_}] :=
          Module[{δ = point - rotPoint, parallel, normal},
                parallel = δ.dir1 dir1;
                normal = Sqrt[#.#]&[δ - parallel];
                rotPoint + c normal dir2 + s normal dir3 + parallel];
         (* rotate points *)
         ℛ[l_] := Module[{dir1, dir2, dir3},
                 (* three orthogonal directions *)
                 dir1 = n[Subtract @@ Take[l, 2]];
                 dir2 = n[(Plus @@ l)/Length[l] - (Plus @@ Take[l, 2])/2];
                 dir3 = -Cross[dir1, dir2];
                 Map[N[r[#, l[[1]], {dir1, dir2, dir3}]]&, l, {-2}]];
         (* prepare lists *)
         allm[l_] := Table[RotateLeft[l, i], {i, Length[l] - 1}];
         argch[l_] := Join[Reverse[Take[l, 2]], Reverse[Drop[l, 2]]];
         (* make a hole in a polygon *)
         makeHole[l_] :=
          With[{mp = (Plus @@ l)/Length[l], h = Append[#, First[#]]&[l]},
                 MapThread[Polygon[Join[#1, Reverse[#2]]]&,
                {Partition[h, 2, 1], Partition[mp + 0.8(# - mp)& /@ h, 2, 1]}]];
         (* wireframe or polygons *)
         makeLine[l_] := Line[Append[l, First[l]]];
         (* show graphics *)
         Show[Graphics3D[If[faceShape === Polygon, makeHole[#], makeLine[#]]& /@
          Join[{Table[N[{Cos[φ], Sin[φ], 0}], {φ, 0, 2Pi - 2Pi/n, 2Pi/n}]},
         (* build layer on layer *)
         If[iter > 0, Flatten[NestList[myUnion[argch /@ (ℛ /@
          Flatten[Join[allm /@ #], 1])]&, Join[argch /@ (ℛ /@ #)]&[(* one face *)
            Table[Table[N[{Cos[φ], Sin[φ], 0}],
                        {φ, φ0, φ0 + 2Pi - 2Pi/n, 2Pi/n}],
                  {φ0, 0, 2Pi - 2Pi/n, 2Pi/n}]], iter - 1], 1], {}]]], opts]]
```

First, let us see how often we have typical list operations (dealing with expressions with head `List`), such as `Map`, `Dot`, `Join`, `Apply`, `Table`, `Flatten`, `Reverse`, `Partition`, `Take`, `Drop`, `MapThread`, `Part`, and `List` itself (all of these functions we will discuss in this chapter) in the source code of `GluedPolygons`.

```
MapThread[{#, Count[#2, #1]}&,
(* the commands to be counted *)
{{List, Map, Dot, Join, Apply, Table, Flatten, Reverse,
  Partition, Take, Drop, MapThread, Part},
 Table[#, {13}]&[ (* the code to be analyzed *)
 Level[DownValues[GluedPolygons], {-1}, Heads -> True]]}]
```

When actually running the code, these operations are carried out more frequently because of loops. With `Trace`, we can look at how often the commands listed above appear in the history of the function evaluation. (Because it is a very simple graphic, we suppress its rendering and have a look at a slightly more complicated example in a moment.)

```
glueTrace =
Trace[GluedPolygons[5, 3Pi/4, 1, Polygon, DisplayFunction -> Identity],
      (* the commands to be counted *)
      Part | Map | Dot | Apply | Flatten | Table | Reverse |
      Partition | Take  | Join | Drop  | MapThread | List];

Function[arg, {#, Count[arg, #]}& /@
(* count how often they appear in glueTrace *)
{List, Reverse, Join, Dot, Map, Partition, Apply, Take,
 MapThread, Drop, Table, Part, Flatten}][
     Level[glueTrace, {-1}, Heads -> True]]
```

(These numbers are not the actual function calls because inside `Trace` they appear hierarchically nested.) `glueTrace` is quite a big object—again, a `List` structure.

```
{ByteCount[glueTrace], Depth[glueTrace], LeafCount[glueTrace]}
```

Now, having "established" the importance of `List` operations, we show two pictures generated with `GluedPoly`⋱ `gons`.

```
Show[GraphicsArray[{
 GluedPolygons[4, 3Pi/4, 4, Line, DisplayFunction -> Identity],
 GluedPolygons[6, 3Pi/4, 2, Polygon, DisplayFunction -> Identity]}]]
```

For certain angles and certain polygons, we just get the regular polyhedra (we do not see the "top" polygons because for the given number of iterations it was not generated).

```
Show[GraphicsArray[{
 GluedPolygons[4, Pi/2, 1, Polygon,
              DisplayFunction -> Identity, Boxed -> False],
 GluedPolygons[5, 2.0344, 2, Polygon,
              DisplayFunction -> Identity, Boxed -> False]}]]
```

In addition to the tetrahedron, the octahedron, and the icosahedron, with triangles, we can form the following polyhedron [280✱], [202✱], [203✱], [48✱].

```
GluedPolygons[3, 0.729729, 4, Polygon, Boxed -> False,
             SphericalRegion -> True, ViewPoint -> {1, 1, 1}]
```

For certain initial polygons and certain angles, many edges coincide and we get interesting polyhedra. Here, two examples for a heptagon and an octagon are shown.

```
Show[GraphicsArray[{
GluedPolygons[7, 53/120 Pi, 2, Polygon, DisplayFunction -> Identity],
GluedPolygons[8, Pi/2, 2, Polygon, DisplayFunction -> Identity]}]]
```

Using an animation (to be discussed in the next chapter), we can see how a dodecahedron forms. In addition to the dodecahedron, we see a second nice polyhedron made from regular pentagons at $\varphi \approx 1.1074$.

```
Show[GraphicsArray[#]]& /@ Partition[
Table[GluedPolygons[5, N[φ], 2, Polygon, Boxed -> False,
              SphericalRegion -> True, DisplayFunction -> Identity],
      {φ, Pi, 0, -Pi/34}], 5]
```

```
Do[GluedPolygons[5, N[φ], 2, Polygon, Boxed -> False,
                 SphericalRegion -> True], {φ, Pi, 0, -Pi/59}]
```

Next, we will fold four rings of regular hexagons. To avoid many intersecting polygons and to better view the inner hexagons we display lines instead of hexagons. For the three folding angles $\pi/2$, $\pi/2 \pm 4\pi/37$ many hexagon edges coincide. The following graphics display the folded hexagons at these angles and at 1% different angles.

```
foldedHexagons[φ_, opts___] :=
Module[{c = 0},
 Show[GluedPolygons[6, N[φ], 3, Line, PlotLabel -> N[φ],
             DisplayFunction -> Identity] /. (* colored edges *)
     l_Line :> {Thickness[0.001], Hue[(c = c + 1)/230], l}, opts]]

Function[φ, Show[GraphicsArray[foldedHexagons[#]& /@
    {0.99 φ, φ, 1.01 φ}]]] /@ {Pi/2 - 4/37 Pi, Pi/2, Pi/2 + 4/37 Pi}
```

The following animation shows the dynamics of the folding process.

```
Do[foldedHexagons[φ, DisplayFunction -> $DisplayFunction],
   {φ, Pi, 0, -Pi/300}];
```

With slight adaptation of the implementation of `GluedPolygons`, it is possible to mirror on vertices and to treat concave polygons (like a pentagram).

Now, we go on to the detailed discussion of the function `List`.

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

6.1 Creating Lists

■ 6.1.1 Creating General Lists

In this subsection, we discuss several ways to create lists. For completeness, we again mention the command `Table`, introduced in Subsection 5.2.2. Note that with `Table`, as with all constructions using analogous iterators (`Sum`, `Product`, `Do`, etc.), the lower and upper limits of the running variables do not have to be numbers; only the difference of the two limits has to be a positive real number greater than the current value of the increment (see Section 4.2).

```
Table[f[i], {i, l[5], l[5] + 6, 1}]
```

`Array` is a somewhat simpler construction.

`Array[`*function*`, {`i_1`, `i_2`, …, `i_n`}]`
produces a "rectangular" list of size $i_1 \times i_2 \times \cdots \times i_n$ with the elements of the form *function*`[`j_1`, `j_2`, …, `j_n`]`, where $1 \leq j_k \leq i_k$.

`Array[`*function*`, {`i_1`, `i_2`, …, `i_n`}, {`$i0_1$`, `$i0_2$`, …, `$i0_n$`}, `*head*`]`

produces a "rectangular" object with the head *head* (instead of a `List`), which at each level has the size $i_1 \times i_2 \times \cdots \times i_n$ and contains the elements *function* `[`j_1`, `j_2`, …, `j_n`]`. The *j*th variable runs from $i0_j$ to $i_j + i0_j$-1.

For a one-dimensional (1D) list (i.e., a vector that does not necessarily have exactly three components), we also have the following construct.

`Range[`i_{min}`, `i_{max}`, `i_{step}`]`

produces a list of the numbers (or more general expressions) between i_{min} and i_{max} with step i_{step}.

Before presenting a few examples, we take note of a recurring theme in this chapter.

Many operations that can be typically carried out on lists (head `List`) or with lists, also work for expressions with other heads.

The following example shows a triply-nested list with individual elements having different lengths, so that the list is not rectangular.

```
Table[fgh[i, j, k], {i, 3}, {j, i}, {k, j}]
```

(`TreeForm` can be used to "see better" the nonrectangular form of smaller examples.) `Array` produces a rectangular-shaped object.

```
Array[fu, {3, 3, 3}]
```

The first argument of `Array` can be any *function*, including a symbol or pure function, of course.

```
Array[Times[#1, #2, #3]&, {3, 3, 3}]
```

Here is the same thing with a shorter input.

```
Array[Times[##]&, {3, 3, 3}]
```

This input is still shorter.

```
Array[Times, {3, 3, 3}]
```

If a fourth argument appears in `Array`, every pair of braces `{}` is replaced by that argument. In the following example, the fourth argument is `H`.

```
Array[fu, {3, 3, 3}, 1, H]
```

In addition to giving objects of rectangular form, `Array` has another distinguishing feature when compared with `Table`: The step size of the dummy variable in `Array` is always 1. The advantage of `Array` compared with `Table` is that an auxiliary variable is not needed, and so localization of variables (as discussed in Subsections 4.6.1 and 4.6.3) is automatically avoided. Note that `Array`, in contrast to `Table`, always needs an integer second argument. Another difference is obvious if we look at the attributes of `Array` and `Table`.

```
{Attributes[Table], Attributes[Array]}
```

Thus, `Table` recomputes its first argument for every call, whereas `Array` does this at the beginning, to the extent possible. We now illustrate these differences and at the same time show that they have a natural effect on the computation times required when using `Table` compared with `Array`. Note the generation of the *i* in `a`*i*.

```
Remove[a, i, j];
a = 0;
Table[a = a + 1; ToExpression[StringJoin["a" <> ToString[a]]][i, j],
      {i, 3}, {j, 3}]
```

```
a = 0;
Array[a = a + 1; ToExpression[StringJoin["a" <> ToString[a]]], {3, 3}]
```

The preevaluation of the first argument makes `Array` much faster than `Table`.

```
Do[a = 0;
Table[a = a + 1; ToExpression[StringJoin["a" <> ToString[a]]][i, j],
      {i, 3}, {j, 3}], {1000}] // Timing
```

```
Do[a = 0;
Array[a = a + 1; ToExpression[ StringJoin["a" <> ToString[a]]],
                                        {3, 3}], {1000}] // Timing
```

For efficiency, expressions (especially large ones) should be at least partially computed whenever the computed expression is “simpler” than the beginning expression. This evaluation can be done via `Table[Evaluate[`*preComputable*`],` *iterators*`]`. We will discuss an application of this kind shortly. Care should be taken not to perform this precomputation when the symbolic result differs from the result after substitution of the dummy variable. Care should also be taken if the symbolic expression evaluates (such as in cases of nested tables, sums, and so on) to large expressions.

`Range` works almost exclusively with numbers (i_{max} and i_{min} have to differ by a numeric constant); the prescribed limits are never exceeded.

```
Range[-3, 4, 0.98]
```

This input generates the reversed list.

```
Range[4, -3, -0.98]
```

Now, the result is the empty list.

```
Range[4, -3, 0.98]
```

In the next example, the difference between the upper and lower limits is a real number greater than the step size 3/2.

```
Range[-3 + chevy, 4 + chevy, 3/2]
```

Here steps along a direction in the complex plane are taken and the endpoint is not in the resulting list.

```
Range[6 + 4 I, -3 - 3 I, -(9 + 7 I)/(12/10)]
```

Note that in all iterator-carrying functions, the generated iterator value depends on the type of limits. In the following examples they are either of type `Real`, `Integer`, or `Complex`.

```
Table[abcd[i], {i, 1, 5, 1}]
```

```
Table[abcd[i], {i, 1.0, 5.0, 1.0}]
```

```
Table[abcd[i], {i, 1.0, 5.0 + I 0.0, 1.0}]
```

```
Table[abcd[i], {i, 1.0 + I 0.0, 5.0, 1.0}]
```

In the following input, be sure to note the first term, whose argument has the head `Integer`; the arguments of the other terms have the head `Real`.

```
Table[abcd[i], {i, 1, 5.0, 1.0}]
```

The iterator steps are calculated in such a way that the last element in the following `Table` has the argument 5 (head `Real`).

```
Table[abcd[i], {i, 1., 5, 1.0}]
```

For a square matrix, the computation times and the required memory grow quadratically with its size, assuming all matrix elements are about the same size and are equally difficult to compute. We now illustrate this for $n \times n$ matrices with `Null` entries. The gray lines in the graphic represent quadratic approximations of the construction time and memory use, respectively. (We discuss the command `Fit` in Chapter 2 of the Numerics volume [302✱].) The measured timings clearly show the finite resolution of the `Timing` command.

```
Module[{datat, datam, approxt, approxm, n = 300},
(* times *)
datat = Array[{#, Timing[Array[Null&, {#, #}];][[1, 1]]}&, {n}];
(* fit to times used *)
approxt = Fit[datat, {1, x^2}, x];
(* memory used *)
datam = Array[{#, ByteCount[Array[Null&, {#, #}]]/1024}&, {n}];
(* fit to memories used *)
approxm = Fit[datam, {1, x^2}, x];
(* the picture *)
Show[GraphicsArray[
ListPlot[#[[1]], #[[3]], PlotRange -> All,
         (* data points as black points *)
         PlotStyle -> {GrayLevel[0], PointSize[0.006]},
         (* fit as underlying gray curve *)
         Prolog -> {GrayLevel[1/2], Thickness[0.01],
                    Line[Table[{x, #[[2]]}, {x, 0, n, 1}]]},
         DisplayFunction -> Identity]& /@
{{datat, approxt, AxesLabel -> {"dim", "t in s"}},
 {datam, approxm, AxesLabel -> {"dim", "Mem. in kByte"}}}]]]
```

So far, we discussed functions to generate lists “from scratch”. Often one has already a *Mathematica* expression and one wants to convert it or parts of it into (nested) lists. Here is an example: Starting with an object with several arguments, we want to use it to make a list, or starting with a list, we want to use its elements as arguments for a function. This transformation of the heads can be accomplished as follows.

```
makeNewHead[oldHead_[arguments__], newHead_] := newHead[arguments]
```

Here is how it works.

```
makeNewHead[funcManyArgs[x1, x2, x3, x4, x5, x6, x7, x8], List]
```

Here it is in reverse.

```
makeNewHead[%, funcManyArgs]
```

This process can be done more easily with `Apply`.

`Apply[`*newHead*`, `*expression*`, `*levelSpecification*`]`
or
if *levelSpecification* is equal to `{0}`, *newHead* `@@` *expression*

if *levelSpecification* is equal to `{1}`, *newHead* `@@@` *expression*

replaces the head of expression at the level *levelSpecification* by *newHead*. If *levelSpecification* is not present, it is assumed to be `{0}`.

Only the head will be replaced; the inner lists remain unchanged.

```
newHead @@ Array[r[##]&, {3, 4}]
```

Now, we apply `newHead` to various levels.

```
Apply[newHead, Array[𝕣, {2, 2}, {2, 4}], {-2}]
```

```
Apply[newHead, Array[𝕣, {2, 2}, {2, 4}], 3]
```

```
Apply[newHead, Array[𝕣, {2, 2}, {2, 4}], Infinity]
```

In the next input, all `List` heads are placed by `newHead` heads.

```
Apply[newHead, Array[𝕣, {2, 2}, {2, 4}], {0, Infinity}]
```

The head of raw expressions does not get changed by `Apply`.

```
Apply[HeadNew, Array[𝕣, {2, 2}, {2, 4}], {-1}]
```

> `Apply` is a very efficient function and should be used often, especially when manipulating larger expressions.

Next, we generate a long list of machine numbers using `Range`.

```
longList = Range[1, 1000000, 1];
```

We compute its sum. Let us compare the timings of various ways to sum the term of `longList`. Because all summands are machine integers *Mathematica* can use internal optimizations to carry out the `Do` loop quickly.

```
Timing[sum = 0;
       Do[sum = sum + longList[[i]], {i, 100000}];
       sum]
```

```
Timing[Apply[Plus, longList]]
```

Now let us sum another list of integers, but not machine integers. `Do` has the attribute `HoldAll`; that is, for every call, the *i*th element of `longList` is looked up and added to `sum`. In contrast, `Apply` works "only once" on the entire object `longList`. This time the `Apply` version is many times faster. This is not unexpected. `Apply[Plus, longList]` can deal with all 10^5 summands at once, while the `Do` loop has to deal with all summands individually.

```
longList = 10^100 Range[1, 10^5];
```

```
Timing[sum = 0;
       Do[sum = sum + longList[[i]], {i, 10^5}];
       sum // N]
```

```
Timing[Apply[Plus, longList] // N]
```

Next, we use a list with symbolic entries. In this example, the timings are nearly the same.

```
(* ξ is a symbol without a value *)
longList = ξ Range[1, 100000, 1];
{Timing[sum = 0;
        Do[sum = sum + longList[[i]], {i, 10000}];
        sum],
Timing[Apply[Plus, longList]]}
```

For more complicated symbolic list entries, the timing difference might be much larger. (In the following example, the timing difference is caused by repeated reordering of `sum` into canonical form after each call to `Plus` in `sum = sum + longList[[i]]`.)

```
Clear[ξ];
longList = Table[ξ^i + ξ^(i + 1), {i, 1000}];
{Timing[sum = 0;
        Do[sum = sum + longList[[i]], {i, 1000}];
        sum;],
Timing[Apply[Plus, longList];]}
```

A time ratio on the order of 10 between procedural and functional programs is typical. Of course, the savings depends on the concrete implementation and the size of the objects involved, but we will see about one order of magnitude ratios in similar computations below.

Using the function `Apply`, we can implement a function `Arguments`. `Arguments[`*expr*`]` returns the sequence of arguments of *expr*. This means *expr* equals `Head[`*expr*`][Arguments[`*expr*`]]`.

```
Arguments[expr_] := Apply[Sequence, Unevaluated[expr]]
```

The head `Sequence` of the result allows for a straightforward application of `Head[`*expr*`]` to the arguments. Here is a simple example.

```
expr = C[3, 4];
Arguments[expr]

Head[expr][Arguments[expr]]
```

The `Unevaluated` on the right-hand side is needed when `Arguments` is called with an unevaluated argument.

```
Arguments[Unevaluated[Plus[1, 1]]]
```

For functions having the attribute `SequenceHold`, the input `Head[`*expr*`][Arguments[`*expr*`]]` will not evaluate to *expr*. Here is an example.

```
expr =  C -> 1;
Head[expr][Arguments[expr]]
```

Evaluating the arguments before applying the head yields the original expression.

```
#1[##2]&[Head[expr], Arguments[expr]]
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

■ 6.1.2 Creating Special Lists

The identity matrix (Kronecker symbol δ_{ij}) and the Levi-Civita tensor ε_{ijk} fall into the category of special matrices (tensors [73★]). In *Mathematica*, the identity matrix is `IdentityMatrix`.

`IdentityMatrix[`*dim*`]`
creates a *dim*-dimensional identity matrix.

Here is the identity matrix of dimension 6.

```
IdentityMatrix[6]
```

An obvious generalization of the identity matrix is the diagonal matrix.

`DiagonalMatrix[`*mainDiagonal*`]`
creates a square matrix with the values contained in the list *mainDiagonal* on the main diagonal and zeros everywhere else.

Here is a diagonal matrix of dimension 8.

```
DiagonalMatrix[Range[8]]
```

The representation of Levi-Civita tensors can be accomplished with the help of `Signature`. It is not a matrix, but it can easily be used to construct one.

`Signature[`*listOfNumbers*`]`

gives 1 if the numbers in *listOfNumbers* are an even permutation of `{1, 2, 3, 4, 5, ... , Length[`*listOfNumbers*`}]`. It gives -1 if they are an odd permutation, and it gives 0 otherwise. (If the elements of *listOfNumbers* are not numbers, the canonical order determines the sign.

Here are the Levi-Civita tensors of dimensions 2 through 4.

```
Table[Signature[{a, b}], {a, 2}, {b, 2}]

Table[Signature[{a, b, c}], {a, 3}, {b, 3}, {c, 3}]

Table[Signature[{a, b, c, d}], {a, 4}, {b, 4}, {c, 4}, {d, 4}]
```

A Levi-Civita tensor of dimension 6 has $6^6 = 46\,656$ entries; 2×360 of these entries are ± 1.

```
{Count[#, 1, {-1}], Count[#, -1, {-1}], Count[#, 0, {-1}]}&[
   Table[Signature[{a, b, c, d, e, f}],
         {a, 6}, {b, 6}, {c, 6}, {d, 6}, {e, 6}, {f, 6}]]
```

If the arguments are not computed to be numbers, their order is determined by the canonical order and so it is decided whether the permutation is even or odd (this canonical sorting, of course, also is the case if the arguments are numbers). If two arguments of `Signature[{...}]` are identical, the result is 0.

```
Clear[asdf, ⅇ];

{Signature[{1, asdf, ⅇ[t]}], Signature[{asdf, 1, ⅇ[t]}],
 Signature[{asdf, 1, ⅇ[t], ⅇ[t]}]}
```

For comparison, the arguments in their canonical order are shown here.

```
SetAttributes[orderlessFunction, Orderless]

{orderlessFunction[1, asdf, ⅇ[t]],
 orderlessFunction[asdf, 1, ⅇ[t]],
 orderlessFunction[asdf, 1, ⅇ[t], ⅇ[t]]}
```

The Levi-Civita tensor is a very important object. It permits a "correct" (also valid for left-hand coordinate systems) notation for the cross product $(\boldsymbol{a} \times \boldsymbol{b})_i = \varepsilon_{i\,j\,k}\, a_j\, b_k$ of two three-element vectors $\boldsymbol{a}$ and $\boldsymbol{b}$, where the right-hand side is to be summed over values of j and k, each ranging from 1 to 3, and c_i is the ith component of the vector $\boldsymbol{c}$.

```
Table[Sum[Signature[{i, j, k}] a[j] b[k], {j, 3}, {k, 3}], {i, 3}]
```

The last result agrees with the known result (`𝕦[i]`, `𝕦[j]`, and `𝕦[k]` represent unit vectors in the following; the function `Collect` collects with respect to the `𝕦[`*ijk*`]` terms; we will discuss it in Chapter 1 of the Symbolics volume [303✱]). The command `Det` gives the determinant; we discuss it soon.

```
Remove[a, b, i, j, k];
Det[{{𝕦[i], 𝕦[j], 𝕦[k]},
     {a[1], a[2], a[3]},
     {b[1], b[2], b[3]}}] // (* rewrite result *) Collect[#, _𝕦]&
```

Assuming that we want to also look at higher dimensional Levi-Civita tensors, we would need to generate the iterator sequence {*a*, *dim*}, {*b*, *dim*}, {*c*, *dim*}, {*d*, *dim*}, …, automatically [56✱].

```
iteratorList[var_String, iMax_Integer] :=
  Table[{ToExpression[var <> ToString[i]], iMax}, {i, iMax}]
```

```
iteratorList["arg", 4]
```

In the following example, we will use a construction of the form Table[*func*, ##]& @@ {*listOfSingleIterators*} to "put in" the different {arg*i*, *j*} without the outermost brackets in the Table. We now look at the result of our iterator construction.

```
Table[{arg1, arg2, arg3}, ##]& @@ iteratorList["arg", 3]
```

Another possibility would be to use Sequence to get rid of the outer brackets: Table[*func*, Evaluate[Se⋱ quence @@ *listOfSingleIterators*]].

```
Table[{arg1, arg2, arg3}, Evaluate[Sequence @@ iteratorList["arg", 3]]]
```

The argument of Signature can be constructed analogously to String.

```
signArg[var_String, iMax_Integer] :=
   Table[ToExpression[var <> ToString[i]], {i, iMax}];

signArg["arg", 6]
```

We can now write a routine that creates several Levi-Civita tensors at once and prints them (using Print).

```
moreLeviCivitaTensors[dimMin_, dimMax_] :=
Module[{iter, siar},
Do[siar = signArg["argu", j];
   iter = iteratorList["argu", j];
   CellPrint[Cell["∘ Levi-Civita tensor of "<> ToString[j] <>
                  ToString[Which[j === 2, "nd", j === 3, "rd",
                                 j >= 4, "th"]] <> " order:", "PrintText"]]
   Print[Table[Signature[siar], ##]& @@ iter], {j, dimMin, dimMax}]]
```

Here are the first three Levi-Civita tensors. (The fifth tensor already has $5^5 = 3125$ elements.)

```
moreLeviCivitaTensors[2, 4]
```

Instead of the symbols a*i* we could, of course, also use nonatomic expressions, such as a[*i*], for the iterators.

Let us give one more example of using multiple iterators. The Stirling numbers of the second kind $\mathcal{S}_n^{(k)}$ (to be discussed in Chapter 2 of the Numerics volume [302✱]) have the following multiple sum representation [59✱], [213✱].

$$\mathcal{S}_n^{(k)} = \frac{n!}{k!} \sum_{r_1=1}^{n} \cdots \sum_{r_k=1}^{n} \left(\frac{\delta_{n, \sum_{j=1}^{k} r_j}}{\prod_{j=1}^{k} r_j!} \right)$$

Using an Evaluate[Sequence @@ *listOfSingleIterators*] construction, we can directly implement stirling⋱ S2[*n*, *k*].

```
stirlingS2[n_Integer?Positive, k_Integer?Positive] :=
Module[{r},
n!/k! Sum[KroneckerDelta[n, Sum[r[j], {j, k}]]*
          1/Product[r[j]!, {j, k}],
          (* the iterator *)
          Evaluate[Sequence @@ Table[{r[i], n}, {i, k}]]]]
```

Here is an example.

```
stirlingS2[8, 5] // Timing
```

`stirlingS2[`*n*`, `*k*`]` works, although slowly. The $\delta_{n,\sum_{j=1}^{k} r_j}$ term results in most summands being zero. Instead of summing all 32768 summands in the above example, it is much more efficient to restrict the summation to the (56 in the last example) values of the iterators to such values that the summand is nonvanishing. As a first step in this direction, we use the condition $n = \sum_{j=1}^{k} r_j$ to restrict the iterator limits.

$$\mathcal{S}_n^{(k)} = \frac{n!}{k!} \sum_{r_1=1}^{n} \sum_{r_2=1}^{n-r_1} \cdots \sum_{r_k=1}^{n-r_1-\cdots-r_{k-1}} \frac{\delta_{n,\sum_{j=1}^{k} r_j}}{\prod_{j=1}^{k} r_j!}$$

The iterators can contain functions of the iterator variables, and so, we can implement the upper limits in the last sum $n - r_1 - \cdots - r_j$ in the following manner.

```
stirlingS2Fast[n_Integer?Positive, k_Integer?Positive] :=
Module[{r},
       n!/k! Sum[KroneckerDelta[n, Sum[r[j], {j, k}]]*
                 1/ Product[r[j]!, {j, k}],
                        Evaluate[Sequence @@
                 Table[{r[i], n - Sum[r[j], {j, i - 1}]}, {i, k}]]]]
```

`stirlingS2Fast` is of course much faster.

```
stirlingS2Fast[8, 5] // Timing
```

Using the identity $n = \sum_{j=1}^{k} r_j$ forced by the arguments of the Kronecker symbol we can eliminate the last iterator r_k.

```
stirlingS2Fastest[n_Integer?Positive, k_Integer?Positive] :=
Module[{r},
       n!/k! Sum[If[(r[k] = n - Sum[r[j], {j, k - 1}]) > 0,
                 1/ Product[r[j]!, {j, k}], 0],
                        Evaluate[Sequence @@
                 Table[{r[i], n - Sum[r[j], {j, i - 1}]}, {i, k - 1}]]]]

stirlingS2Fastest[8, 5] // Timing
```

This yields another slight timing improvement for larger *k* and *n*.

```
stirlingS2Fast[16, 8] // Timing

stirlingS2Fastest[16, 8] // Timing
```

All here implemented Stirling number calculating functions `stirlingS2`, `stirlingS2Fast`, and `stirlingS2`↘
`Fastest` agree with the built-in `StirlingS2`.

```
StirlingS2[8, 5] // Timing
```

Sometimes we need an "indexed version" of a unit tensor. While `Signature` generates a completely antisymmetric tensor containing 0s and 1s, the function `KroneckerDelta` generates a unit diagonal tensor.

`KroneckerDelta[`*sequenceOfNumbers*`]`
gives 1 if the numbers in *sequenceOfNumbers* are all identical and 0 else.

Here are the values of `KroneckerDelta` for 1, 2, and 3 arguments.

```
Table[KroneckerDelta[i], {i, -2, 2}] // TableForm

Table[KroneckerDelta[i, j], {i, -2, 2}, {j, -2, 2}] //
               TableForm[#, TableSpacing -> {1, 1}]&
```

```
Table[KroneckerDelta[i, j, k], {i, -2, 2}, {j, -2, 2}, {k, -2, 2}] //
              TableForm[#, TableSpacing -> {1, 1}]&
```

`KroneckerDelta` uses `Equal` for comparison. So the following input returns 1.

```
KroneckerDelta[0.9999999999999999999, 1]
```

The following two low-precision numbers are equal.

```
KroneckerDelta[4.`0*^-20, -2.`0*^-20]
```

The equality of $(\pi + 1)^2$ and $\pi^2 + 2\pi + 1$ cannot be established numerically. So the following input does not evaluate to 1.

```
KroneckerDelta[(Pi + 1)^2, Expand[(Pi + 1)^2]]
```

For symbolic arguments, `KroneckerDelta` stays unevaluated.

```
KroneckerDelta[αβγ, abc]
```

Often, the function `KroneckerDelta` is used inside `Sum`. In the following infinite sum, the 1000th element is selected.

```
Sum[KroneckerDelta[k, 1000] ϕ[k], {k, Infinity}]
```

Two further special sets of new (nested) lists that one occasionally wants to create from a given list are subsets and tuples.

`Subsets[`*list*`, {`*length*`}]`

gives a list of all length *length* subsets of the elements from the list *list*. If the second argument is absent, all subsets are returned.

Here are all the subsets of the list {1, 2, 3, 4, 5} shown.

```
With[{𝕝 = {1, 2, 3, 4, 5}},
Do[CellPrint[Cell[TextData[{"∘ The length " <> ToString[k] <>
                   " subsets of " <>  ToString[𝕝] <> " are: ",
                   Cell[BoxData[FormBox[StyleBox[
                    MakeBoxes[#, StandardForm]& @ Subsets[𝕝, {k}], "MR"],
                                  StandardForm]]]}]]], {k, 0, Length[𝕝] + 1}]
```

In general, a list of length k has 2^k subsets (including the empty set and the whole list).

```
Table[{k, Length[Subsets[Range[k]]]}, {k, 12}]
```

In the subsets, each list elements occurs exactly once. If we allow for repetitions, we get tuples.

`Tuples[`*list*`, {`*length*`}]`

gives a list of all length *length* subsets of the elements from the list *list*. If the second argument is absent, all subsets are returned.

Here are all 1, 2, 3, and 4-tuples formed from the list {1, 2, 3}.

```
With[{𝕝 = {1, 2, 3}},
Do[CellPrint[Cell[TextData[{"∘ The length " <> ToString[k] <>
                   " tuples of " <>  ToString[𝕝] <> " are: ",
                   Cell[BoxData[FormBox[StyleBox[
                    MakeBoxes[#, StandardForm]& @ Tuples[𝕝, {k}], "MR"],
                                  StandardForm]]]}]]], {k, 0, Length[𝕝] + 1}]
```

The number of k-tuples formed from a list of length j is j^k.

```
Table[{k, Length[Tuples[Range[j], k]]}, {j, 6}, {k, 6}]
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

6.2 Representation of Lists

Lists are represented with `{...}` or internally via `List[...]`. If the depths of the list structure are two, they are often easier to view as two-dimensional (2D) matrices.

`MatrixForm[`*matrix*`,` *options*`]`

formats the “rectangular” object *matrix*, using the options *options*.

`TableForm[`*list*`,` *options*`]`

formats the list *list* “in tabular form”, using the options *options*. Here, *list* does not have to be an object of “rectangular form”.

The possible options for these two commands are the following.

```
Options[MatrixForm]
```

```
Options[TableForm]
```

Here are the options listed individually (the meanings of the concepts of `Column`, `Row`, `Center`, `Left`, `Right`, `Bottom`, and `Top` should be obvious).

`TableDirections`

defines the direction (horizontal or vertical) in the consecutive dimensions.

Default:

`Automatic` (=`{Row, Column, … }`)

Admissible:

`{`*rowOrColumn*$_1$`,` *rowOrColumn*$_2$`, … }` with

rowOrColumn$_i$ = `Column` or *rowOrColumn*$_i$ =`Row`

`TableDepth`

defines the maximum number of directions for the table to be printed.

Default:

`Infinity`

Admissible:

`1, 2, 3, … , Infinity`

`TableHeadings`

defines the labels for the directions to be printed.

Default:

`None`

Admissible:

{*heading*$_1$, *heading*$_2$, …, *heading*$_n$}

`TableSpacing`

defines the amount of space between rows and columns in the directions to be printed.

Default:

`Automatic` (=`{1, 1, 1, … }`)

Admissible:

{*integer*$_1$, *integer*$_2$, …, *integer*$_n$}

`TableAlignments`

defines the centering of the *i*th dimension.

Default:

`Automatic`

Admissible:

{*lbrct*, *lbrct*, … } with *lbrct* ∈ `{Left, Bottom, Right, Center, Top}`

The following example creates a triply-nested list `ttt`.

```
ttt = Table[f[a, b, c], {a, 3}, {b, 2}, {c, 3}]
```

Here is a somewhat more readable display of `ttt` produced using `TableForm[…]`. The first "dimension" should be regarded as a column, the second "dimension" as a row, and the third "dimension" again as a column.

```
TableForm[ttt, TableDirections -> {Column, Row, Column},
               TableSpacing -> {4, 3, 2},
               TableAlignments -> {Center, Bottom, Right},
               TableHeadings ->
                ({{"OuterColumn[1]", "OuterColumn[2]", "OuterColumn[3]"},
                  {"MiddleRow[1]", "MiddleRow[2]"},
                  {"InnerColumn[1]", "InnerColumn[2]", "InnerColumn[3]"}} /
                  (* headers in bold *)
                  (Map[StyleForm[#, FontWeight -> "Bold"]&, #, {-1}]&))]
```

The headings are not shown in `MatrixForm`.

```
MatrixForm[%]
```

We give no additional explicit examples here; we will make frequent use of `TableForm` along with its options in this and later chapters.

Lists can be arbitrarily deeply nested. A very important special case of a nested list is a tensor. A tensor is a "rectangular" array of $n_1 \times n_2 \times \cdots \times n_k$ expressions. Here, k is called the tensor rank. It can be determined by using the function `TensorRank`.

`TensorRank[`*nestedList*`]`

gives the maximal depths such that *nestedList* is a tensor.

Here are four simple examples.

```
TensorRank[2]

TensorRank[{2}]

TensorRank[{{2, 2}, {2, 2}}]
```

```
TensorRank[{{{{{2}}}}}]
```

In the next input, one element is missing in the {3, 3} element and so the resulting tensor rank is 2.

```
{{{ 1,  2,  3}, { 2,  4,  6}, { 3,  6,  9}},
 {{ 2,  4,  6}, { 4,  8, 12}, { 6, 12, 18}},
 {{ 3,  6,  9}, { 6, 12, 18}, { 9, 18    }}} // TensorRank
```

In the next input, one element is a list. But the whole expression is still a tensor of rank 3.

```
{{{ 1,  2,  3}, { 2,  4,  6}, { 3,  6,  9}},
 {{ 2,  4,  6}, { 4,  8, 12}, { 6, 12, 18}},
 {{ 3,  6,  9}, { 6, 12, 18}, { 9, 18, {1, 1}}}} // TensorRank
```

Σ (* session summary *) **TMGBs`PrintSessionSummary[]**

6.3 Manipulations on Single Lists

■ 6.3.1 Shortening Lists

A variety of operations can be performed on (potentially nested) lists. One useful command is `Take`.

`Take[`*list*`, `*n*`]`

extracts the first *n* elements of the list *list*.

`Take[`*list*`, -`*n*`]`

extracts the last *n* elements of the list *list*.

`Take[`*list*`, {`*n*`, `*m*`}]`

extracts the *n*th to *m*th elements of the list *list*.

`Take[`*list*`, {`*n*`, `*m*`, `*step*`}]`

extracts the *n*th to *m*th elements in steps *steps* of the list *list*.

`Select[`*list*`, `*criterion*`, `*levelSpecification*`, `*n*`]`

extracts the first *n* elements of the list *list* which satisfy *criterion* from level(s) *levelSpecification*. Satisfy means that *criterion*`[`*element*`]` yields `True`.

`Cases[`*list*`, `*pattern*`, `*levelSpecification*`]`

extracts those elements of the list *list* from level(s) *levelSpecification*, which match the pattern *pattern*.

We have already used `Part[`*list*`, `*n*`]` or `Part[`*list*`, `*partList*`]` or `Part[`*list*`, All]` to extract elements from a list *list*. The main difference from the command `Take` is that with the exception of `All`, the second argument of `Part` needs a complete listing of all elements to be taken, whereas `Take` will allow a much more concise way for taking out many elements. (A further command for extracting parts is `Extract`. Because its functionality is basically the same as the one of `Part`, we will not use it later.) Here, we take out various parts from the list `{1, 2, 3, 4, 5, 6}`.

```
Table[Take[{1, 2, 3, 4, 5, 6}, i], {i, -3, 3}]
```

```
Take[{1, 2, 3, 4, 5, 6}, All]
```

Here is a 5×5 matrix.

```
mat = Array[a, {5, 5}]
```

This input takes out all odd-numbered rows and columns.

```
Take[mat, {1, 5, 2}, {1, 5, 2}] // MatrixForm
```

And this example takes out all even-numbered rows and columns.

```
Take[mat, {2, 5, 2}, {2, 5, 2}] // MatrixForm
```

Here are some simple examples for `Select` and `Cases`.

```
Select[{1, 2, 3, 4, 5, 6}, 2 < # < 5&]

Cases[{1, 2, 3, 4, 5, 6}, _?(2 < # < 5&)]
```

Sometimes one has a list and wants to extract elements according to elements of another list.

`Pick[`*list*`,` *selectList*`,` *pattern*`]`

gives the elements of list that occur at the positions of the list *selectList* that match the pattern *pattern*.

Here is a simple example. The elements returned occur at positions such that the second element has even numbers at these positions.

```
Pick[{1, 2, 3, 4, 5, 6, 7, 8, 9},
     {9, 8, 7, 6, 5, 4, 3, 2, 1}, _?EvenQ]
```

We also have the following commands `First` and `Last`, which give pieces of a list (but only the elements not wrapped in `List`).

`First[`*list*`]`

gives the first element of the first level of *list*. `First[`*expression*`]` is identical to *expression*`[[1]]`, and it is applicable to expressions whose head is not `List`.

`Last[`*list*`]`

gives the last element of the first level of *list*. `Last[`*expression*`]` is identical to *expression*`[[-1]]`, and it is applicable to expressions whose head is not `List`.

Here is a very simple example.

```
First[{1, 2, 3}]
```

Here is another simple example.

```
Last[{1, 2, 3}]
```

Frequently one has to drop the first or last element of a list. (For instance in the results of `FoldList[Plus, 0, ...]` and `FixedPointList[`*f*`, ...]`.) The functions `Most` and `Rest` do this operation.

`Most[`*list*`]`

deletes the last element of the first level of *list*. `Most` is also applicable to expressions whose head is not `List`.

`Rest[`*list*`]`

deletes the first element of the first level of *list*. `Rest` is also applicable to expressions whose head is not `List`.

Here are two simple examples.

```
Most[{1, 2, 3, 4, 5}]
```

```
Rest[{1, 2, 3, 4, 5}]
```

A single command does not exist with respect to the last element. But the function `Drop` allows us to eliminate the last element easily, although only in a two-argument call.

`Drop[`*list*`, (-)`*n*`]`

gives the list *list* with the first (or last) *n* elements removed. Here, the head of list need not be `List`.

`Drop[`*list*`, {`*n*`, `*m*`}]`

gives a list *list* with the elements *n* through *m* removed. Here, the head of list need not be `List`.

`Delete[`*list*`, (-)`*n*`]`

gives a list *list* with the *n*th term (counting from the end) removed. The head of *list* need not be `List`, and *n* can also be a list of positions in the sense of `Position`.

`DeleteCases[`*list*`, `*pattern*`, `*levelSpecification*`]`

gives a list *list* with all elements matching the pattern *pattern* at the level *level* removed. If the third argument is not explicitly given, *levelSpecification* is assumed to be `{1}`, else it acts at the level(s) *levelSpecification*. The head of *list* need not be `List`.

`DeleteCases[`*list*`, `*pattern*`, `*levelSpecification*`, Heads -> True]`

also removes heads.

`Union[`*list*`]`

gives a list *list* with all elements that appear more than once removed. The head of *list* need not be `List`.

The following examples illustrate the effect of `Delete` and `Union`.

```
Delete[{1, 2, 3, 4, 5, 6, 7, 8, 9}, 4]
```

```
Union[{1, 2, 2, 3, 3, 3, 4, 4, 4, 4}]
```

`DeleteCases` is also a very important command in a lot of applications. Here, all products of `I` with anything or with `I` are deleted from a list. Note that `8I` is a complex number and not a product.

```
DeleteCases[{2, 4, I, E, 8 I, i t, It, I t, I I}, _. I]
```

If nothing remains after the deletion process, the result is `Sequence[]`.

```
DeleteCases[1, 1, {0}]
```

With a third argument in `DeleteCases`, we can operate also on inside expressions.

```
DeleteCases[𝕘[2, 4, I, E, 8 I, Null, 𝕘[I, 4I, hj I], i t, It, I t],
              _.I, {2}]
```

With the option `Heads -> True`, we can also work on heads, in which case, only `Sequence`-objects remain in general.

```
(* here nothing is deleted; List only appears as a head *)
DeleteCases[Array[1&, {2, 2, 2, 2}], List, {0, Infinity}]

(* now all heads disappear *)
DeleteCases[Array[1&, {2, 2, 2, 2}], List, {0, Infinity}, Heads -> True]

(* again all heads disappear *)
DeleteCases[Array[1&, {2, 2, 2, 2}], List, {-1}, Heads -> True]
```

For an empty list `{}`, `First`, `Last`, `Take`, and `Part` generate an error message.

```
First[{}]

Last[{}]

Rest[{}]

Take[{}, 1]

Part[{}, 1]
```

With an empty list as an argument, `Union` produces the same empty list. Here, no message is generated.

```
Union[{}]
```

Here is an interesting application of repeatedly shortening a list. We start with the list {1, 2, …, n}. We delete every second element of this list. From the resulting list, we delete very third element, from the resulting list every fourth element, …. The function `delete` deletes every kth element from the list l.

```
delete[l_, k_] := Delete[l, Table[{j}, {j, k, Length[l], k}]]
```

Using the function `FixedPointList`, we iterate the process of eliminating elements. Here, we start with the first 20 integers.

```
FixedPointList[{(* increment counter for the elements to be taken out *)
                 #[[1]] + 1, (* take out the elements *)
                 delete[#[[2]], #[[1]]]}&,
                {2, Range[20]}, SameTest -> (#1[[2]] === #2[[2]]&)];

Last[Transpose[%]] // (TableForm[#, TableSpacing -> 0.6,
                                 TableAlignments -> Right]&)
```

Only some of the primes remain. The next graphic shows the process of taking out. This time, we start with the first 20000 integers.

```
FixedPointList[{#[[1]] + 1, delete[#[[2]], #[[1]]]}&, {2, Range[20000]},
                SameTest -> (#1[[2]] === #2[[2]]&)];

Show[Graphics[{PointSize[0.002], MapIndexed[Point[{#1, #2[[1]]}]&,
                Last /@ %, {2}]}]];
```

When we start with the first n integers, for large n, we are left with $2\big/\pi^{1/2}\sqrt{n}$ integers [12★]. Here, we start with 100000 integers and show the integers that are left after applying the described deletion process. The red curve is the function $f(n) = 2\big/\pi^{1/2}\sqrt{n}$.

```
ListPlot[MapIndexed[{#1, #2[[1]]}&,
         FixedPoint[{#[[1]] + 1, delete[#[[2]], #[[1]]]}&,
                    {2, Range[100000]},
                    SameTest -> (#1[[2]] === #2[[2]]&)][[2]]],
         Prolog -> {Thickness[0.01], Hue[0],
                   Line[Table[{x, 2/Sqrt[Pi] Sqrt[x]},
                        {x, 0, 10^6, 10^3}] // N]},
         PlotStyle -> {GrayLevel[0], PointSize[0.001]}]
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

■ 6.3.2 Extending Lists

`Prepend` adds terms to a given list.

`Prepend[`*list*`,` *newFirstElement*`]`

adds the expression *newFirstElement* to the beginning of the list *list*. The head of *list* need not be `List`.

`Append[`*list*`,` *newLastElement*`]`

adds the expression *newLastElement* to the end of the list *list*. The head of *list* need not be `List`.

`Insert[`*list*`,` *middleElement*`,` *n*`]`

puts the expression *middleElement* into the list *list* at the *n*th position. The head of *list* need not be `List`, and *n* can also be a list of positions in the sense of `Position`.

`Insert[`*list*`,` *middleElement*`,` −*n*`]`

puts the expression *middleElement* into the list *list* at the *n*th position counting from the end.

Here is an example of `Insert`.

```
Insert[list[78, 45], 89, 1]
```

To add something to a named list, we proceed as follows.

```
myList = {1, 2, 3, e, r, t, {8, 9}, 0}

myList = Append[Prepend[myList, BEGINNING], END]
```

The addition of elements to named lists can be done more easily with `PrependTo`.

`PrependTo[`*symbolWithListValue*`,` *newFirstElement*`]`

puts the expression *newFirstElement* at the beginning of the evaluated form of *symbolWithList*·*Value* and names the resulting object again *symbolWithListValue*. The head of the evaluated form of *symbolWithListValue* need not be `List`.

`AppendTo[`*symbolWithListValue*`,` *newLastElement*`]`

adds the expression *newLastElement* at the end of the evaluated form of *symbolWithListValue* and names the resulting object again *symbolWithListValue*. The head of the evaluated form of *symbolWithListValue* need not be `List`.

Now, we add the element `NEWEND` to the list `myList`.

```
AppendTo[myList, NEWEND];
```

```
myList
```

List operating commands are typical commands in which the infix form of operations can be (and is) used. The following operation is an example.

```
myList ~ AppendTo ~ ALLNEWEND
```

The operations `Append`, `Prepend`, `AppendTo`, `PrependTo`, and `Insert` require that the object to be manipulated is a list or an expression with arguments.

```
Append[5, 4]
```

```
Append[trfhcn, 4]
```

This input works.

```
Prepend[list[78, 45, 56], 89]
```

`AppendTo` and `PrependTo` need a named list-like object or they cannot add anything.

```
Remove[l];
AppendTo[l, 34]
```

Note that the following example does not work because the first argument of `PrependTo` is not the name of a list (or other container).

```
PrependTo[AppendTo[myList, NEWEND1], NEWBEGIN];
myList
```

The inner `AppendTo` added `NEWEND1` to `myList`. But the result of this operation was the new value of `myList` and `PrependTo` expects a symbol in its first argument that evaluates to a list. (To accomplish this feature, `AppendTo` and `PrependTo` have the attribute `HoldFirst`.)

```
Attributes[AppendTo]
```

`Append` does not have the `HoldFirst` attribute.

```
Attributes[Append]
```

The first element of `AppendTo` and `PrependTo` has to be an expression that evaluates to an expression with depth greater than zero.

```
𝕝[1] = {1, 2, 3}; AppendTo[𝕝[1], 4]
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

■ 6.3.3 Sorting and Manipulating Elements

A list can be quickly reversed or its elements can be rotated cyclically.

`Reverse[`*list*`]`

gives *list* in reverse order. The head of *list* need not be `List`.

```
Reverse[headNotList[4, 5, 6, 7, 8, 9]]
```

`RotateRight[`*list*`, `*n*`]`

cyclically rotates the elements in the list *list* *n* times to the right. The head of `list` need not be `List`.

`RotateRight[`*list*`, {`n_1`, `n_2`, …, `n_i`}]`

cyclically rotates the elements in the nested list *list* by n_1 in level 1, n_2 in level 2, … to the right.

`RotateLeft[`*list*`, `*n*`]`

cyclically rotates the elements in the list *list* *n* times to the left. The head of *list* need not be `List`.

`RotateLeft[`*list*`, {`n_1`, `n_2`, …, `n_i`}]`

cyclically rotates the elements in the nested list *list* by n_1 in level 1, n_2 in level 2, … to the left.

Here are two small examples.

```
RotateRight[{"3", "r", "o", "t", "a", "t", "e", " ",
                    "r", "i", "g", "h", "t"}, 3]

RotateLeft[NL["1", "r", "o", "t", "a", "t", "e", " ",
                    "l", "e", "f", "t"], 1]
```

This is a somewhat more complicated example. As long as the argument of `f` is not in the canonical order, the argument is cyclically rotated to the right.

```
f[x_?(!OrderedQ[#]&)] := f[RotateRight[x, 1]];
f[x_?OrderedQ] := x

f[{3, 4, 1, 2}]
```

We can follow the steps using `Trace`.

```
Trace[f[{3, 4, 1, 2}]]
```

For the starting list `{1, 3, 2, 4}`, a problem exists because the cyclical rotation never stops. None of the four possible orders represents a list that is ordered according to `OrderedQ`.

```
f[{1, 3, 2, 4}]
```

Sorting in the usual sense can be accomplished with `Sort`.

`Sort[`*list*`, `*sortOrder*`]`

sorts a list according to the comparison function *sortOrder*. If no *sortOrder* is explicitly prescribed, the canonical order (numbers before symbols) is used. The head of *list* need not be `List`. Here, *sortOrder* must be a (pure) function of two arguments, which gives `True` or `False`.

`Sort` is a very important and also quite interesting function. So, we will discuss it in greater detail. Numbers are sorted by size, and letters are sorted alphabetically.

```
Sort[{3, 78, 9, u io, 𝕞, {89}}]
```

Complex numbers are sorted first by ascending order of real parts, and then by ascending order of absolute values of the imaginary parts. This sort order means that complex conjugate numbers come in pairs. The sort order prescription is applied until *sortOrder* produces `True` for all neighboring pairs of elements.

```
Sort[{1 - I, 1 + I, 1 + I/2, 1 - I/2, 2 + I, 2 - I, 0.8 + 0.9 I,
      0.8 - 0.9I, 1 - I/4, 1 + I/4}]
```

A real number is treated as an imaginary number with a vanishing imaginary part.

```
Sort[{1 - I, 1 + I, 1 + I/2, 1 - I/2, 2 + I, 2 - I, 0.8 + 0.9 I,
      0.8 - 0.9I, 1 - I/4, 1 + I/4, -0.9, 1.7, 1}]
```

Here, we sort the numbers 1 to 10 in descending order using *sortOrder* as a pure function. (Just `GreaterEqual` would, of course, give the same result.)

```
Sort[{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}, (#1 >= #2)&]
```

Note that numeric expressions (meaning `NumberQ` would return `True`) are not sorted by size.

```
Sort[{Sqrt[2], Pi, 3, -10 - E}]
```

Using an explicitly specified ordering function allows us to order by size.

```
Sort[{Sqrt[2], Pi, 3, -10 - E}, Less]
```

In the next example, the sorting is alphabetical and not by size, because `Unevaluated` is used. First, the variables `a`, `b`, and `c` are sorted, and then they evaluate to `3`, `2`, and `1`.

```
a = 3; b = 2; c = 1;
Sort[Unevaluated[{b, a, c}]]
```

```
Sort[{1, 2, 3}]
```

This process can be nicely observed with `On[]`.

```
On[]; Sort[Unevaluated[{b, a, c}]]; Off[];
```

If no pair of neighboring elements in the second argument of `Sort` gives a value of `True`, the list remains unchanged. No messages are generated.

```
Sort[{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}, false]
```

Here, all comparisons return `False`.

```
Sort[{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}, False&]
```

`Sort` compares only neighboring elements at every stage. Thus, the following code, which sorts the numbers 1, 2, 3, 4, 5 so that the absolute value of the difference between any two neighbors is greater than 1, does not work as desired.

```
Sort[{1, 2, 3, 4, 5}, Abs[#1 - #2] > 1&]
```

The following sequence of numbers is already in the “right order”, and `Sort` leaves them in their given order.

```
Sort[{1, 3, 5, 2, 4}, Abs[#1 - #2] > 1&]
```

To understand the strategy of `Sort`, we can collect the pairs being compared in each step into a list `collection` by appending the just-compared numbers in the form of a list at the end of `collection`.

```
collection = {};
Sort[{9, 9, 8, 7, 6, 5, 4, 3, 2, 1, 5},
     (AppendTo[collection, {##}]; Greater[##])&]
```

Here is the current value of `collection`.

```
collection
```

To conclude our discussion of `Sort`, we first present a plot of the number of pairs that are compared, assuming the case in which the test always has the truth value `False`.

```
sortLong[i_] :=
Module[{counter = 0},
       Sort[Table[j, {j, i, 0, -1}], (* count comparisons *)
            Function[{x, y}, counter = counter + 1; #]&[
                                                 False]]; counter]
```

Here, the normal ordering is investigated.

```
sortShort[i_] :=
Module[{counter = 0},
       Sort[Table[j, {j, 0, i}], (* count comparisons *)
                  Function[{x, y}, counter = counter + 1; #]&[
                                                   Less]]; counter]
```

The case in which the elements are already in the correct order (right-hand plot) requires considerably fewer comparisons.

```
Show[GraphicsArray[{
(* using sortLong *)
ListPlot[Table[sortLong[i], {i, 2, 70}],
         DisplayFunction -> Identity, PlotLabel -> "unordered list"],
(* using sortShort *)
ListPlot[Table[sortShort[i], {i, 2, 70}],
         DisplayFunction -> Identity, PlotLabel -> "ordered list"]}]]
```

Note that `sortLong` does not produce the worst case scenario.

```
{sortLong[12], sortShort[12],
Module[{i = 0}, Sort[Range[12], (i = i + 1; OddQ[i])&]; i]}
```

We can also monitor the elements that `Sort` compares. In the following example, we start with the list $3^i \bmod 257,\ 1 \leq i \leq 256$ and display the difference of the compared elements as a function of the number of the comparison.

```
Module[{bag = {}, p = 257},
       Sort[Array[PowerMod[3, #, p]&, p - 1, 0],
            (AppendTo[bag, {##}]; Greater[##])&];
        ListPlot[Apply[Subtract, bag, {1}],
                 Frame -> True, PlotRange -> All, Axes -> False,
                 PlotStyle -> {PointSize[0.001]}]]
```

By watching which elements are compared by `Sort`, we can infer its algorithm. Here is a nearly "anti-ordered" list of 100 integers.

```
data = RotateRight[Range[100], 90];
```

We sort the list and keep track of the compared elements.

```
bag = {};
Sort[data, (AppendTo[bag, {##}]; Greater[##])&];
```

The hierarchical structure of the compared elements shows that a mergesort algorithm [271✱] was used.

```
Show[Graphics[
MapIndexed[Rectangle[{#2[[1]], #1} - 1/2,
                     {#2[[1]], #1} + 1/2]&, bag, {2}]],
              Frame -> True, PlotRange -> All, AspectRatio -> 1/2]
```

Here, we show the superposition of the compared elements of 100 unordered lists. We use one color per list.

```
Module[{bag},
Show[Graphics[Table[bag = {};
  Sort[(* the list to be sorted *)
       RotateRight[Range[100], k],
        (AppendTo[bag, {##}]; Greater[##])&];
  {Hue[k/120], MapIndexed[Rectangle[{#2[[1]], #1} - 1/2,
                                    {#2[[1]], #1} + 1/2]&,
                          bag, {2}]}, {k, 0, 100}]],
     Frame -> True, PlotRange -> All, AspectRatio -> 1/2]]
```

We could go on and investigate the complexity of the search algorithm. Overall, we expect an $n\log(n)$ complexity (more precisely, we have the complexity $n\lfloor\log(n)\rfloor + 2n - 2^{\lfloor\log(n)\rfloor+1}$ [271✱]). Next, we take 5000 lists of length 100, each being a random permutation of the integers 1 to 100. The number of comparisons for the list is highly peaked around 564 with a small deviation only. (We will discuss the use of `Compile` in `randomPermutation` in Chapter 1 of the Numerics volume [302✱].)

```
(* fast code to generate a random permutation *)
randomPermutation =
Compile[{{l, _Integer, 1}},
Module[{lTemp = l, λ = Length[l], tmp1, tmp2},
       Do[tmp1 = lTemp[[i]];
          j = Random[Integer, {i, λ}];
          {lTemp[[i]], lTemp[[j]]} = {lTemp[[j]], tmp1},
          {i, Length[l]}];
       lTemp]];

Module[{k}, (* make graphics *)
ListPlot[{#[[1]], Length[#]}& /@ Split[Sort[Table[k = 0;
          Sort[randomPermutation[Range[100]],
              (k = k + 1; Greater[##])&]; k, {5000}]]],
         PlotRange -> All, PlotStyle -> {PointSize[0.01]},
         Frame -> True, Axes -> False]]
```

For more on sorting, see [165✱]; for a detailed analysis of mergesort, see [196✱]; for achieving the minimal number $\lceil\log_2 n!\rceil$ of comparisons, see [238✱].

The following commands are closely related to `Sort`.

`Max[`*list*`]`

gives the largest element of the list *list*.

`Min[`*list*`]`

gives the smallest element of the list *list*.

Here is a simple example.

```
Max[{1, 2, 3, 445689}]
```

`Max` and `Min` use numerical techniques to determine the largest and smallest elements. When the numerical techniques are unable to make a decision, a message is generated and all possible candidates are kept. We will discuss the message `N::meprec` in detail in Chapter 1 of the Numerics volume [302✱].

```
Max[{Sqrt[(2 - Sqrt[2 + Sqrt[2]])/(2 + Sqrt[2 + Sqrt[2]])],
     Tan[Pi/16], notANumericQuantity}]
```

Complex numbers cannot be compared.

```
Min[{I, -I}]
```

By definition, we have the following behavior for empty lists.

```
{Min[{}], Max[{}]}
```

Another operation closely related to sorting lists is splitting a list into sublists.

`Split[`*list*`,` *comparisonFunction*`]`

splits the list *list* into sublists of consecutive “equal” elements. Two elements $element_1$ and $element_2$ are considered equal when the function *comparisonFunction*`[`$element_1$`,` $element_2$`]` yields `True`. When *comparisonFunction* is not present, equality is determined using `SameQ`. The head of *list* need not be `List`.

Here is a straightforward example for `Split`.

```
Split[{1, 2, 2, 3, 3, 3, 4, 4, 4, 4, 5, 5, 5, 5, 5}]
```

`Split` does not sort its argument.

```
Split[{1, 2, 1, 2, 1, 2, 1, 2}]
```

The function `Split` can be used to efficiently count elements of lists. The function `countDifferent`⋱ `Elements[`*l*`]` returns a list of sublists of length two. Each sublist has the form `{`*element*`,` *elementCount*`}`.

```
countDifferentElements[l_List] :=
                    Apply[{#, Length[{##}]}&, Split[Sort[l]], {1}]

countDifferentElements[{1, 3, 4, 2, 5, 4, 3, 3, 3, 2, 4, 2, 1, 2, 3}]
```

Here is a longer list of “random” integers.

```
longList = Table[IntegerPart[100. Sin[k]], {k, 10^5}];
```

Because `countDifferentElements` traverses the list only three times (one time for `Sort`, one time for `Split` and one time for `Apply`; for the already shorter list of sublists of identical elements) it is much faster than counting the frequency of each number separately.

```
Timing[𝕝1 = countDifferentElements[longList];]

Timing[𝕝2 = Table[{j, Length[Cases[longList, j]]},
                  {j, -100, 100}] /. {_, 0, n___} -> n;]
```

The two calculated list are identical.

```
𝕝1 === 𝕝2
```

To apply a function that does not carry the attribute `Listable` (e.g., the pure function `#^2&`) to a list, or if we want to apply any function to a particular level of a list, we use `Map`.

`Map[`*function*`,` *list*`,` *levelSpecification*`]`

or

`Map[`*function*`,` *list*`]`

if *levelSpecification* = `{1}`

or

function `/@` *list*

if *levelSpecification* = `{1}` applies the function *function* to all elements in the list *list* according to *levelSpecification*. The head of *list* need not be `List`.

Here, every element of the list is to be squared.

```
#^2& /@ {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11}
```

Because of the attribute `Listable` of `Power`, the example above could also have been done more easily with `Range`.

```
Attributes[Power]
```

```
Range[11]^2
```

Next, we raise all arguments in 𝕟 to a power.

```
#^(1/nm)& /@ 𝕟[1, 2, 3, 4, 5, a, b, c, d, e]
```

Compare the last result with the pure application of `Power`.

```
𝕟[1, 2, 3, 4, 5, a, b, c, d, e]^(1/nm)
```

> `Map` is one of the most important *Mathematica* commands. We will make heavy use of it starting now.

For the sake of readability, it is often convenient to use nested `Heads`. The following example uses a two-argument function.

```
𝕗[x_, y_] := x + y
```

```
𝕗[1, #]& /@ {1, 2, 3}
```

Next, we use a function that has a nested head.

```
𝕗[x_][y_] := x + y
```

```
𝕗[1][#]& /@ {1, 2, 3}
```

The `#` can be eliminated by mapping just the head `𝕗[1]`.

```
𝕗[1] /@ {1, 2, 3}
```

Note that for symbols, we could use the attribute `Listable`, but `𝕗[1]` cannot have attributes.

```
SetAttributes[𝕗[1], Listable]
```

Using a pure function with an attribute, we can mimic `Map`, but only at level `{1}`.

```
Function[x, 𝕗[1], {Listable}][{1, 2, 3}]
```

Frequently, the elements of lists are subjected to certain transformations, and then the head `List` is to be changed. For example, the elements in the following list are to be squared and then summed.

```
myList = {1, 5, 9};
Apply[Plus, Map[Function[argu, argu^2], myList]]
```

In writing the last input in the short form of *Mathematica* commands, note the following rule.

> `Apply` and `Map` group from the right.

This rule means that parentheses have to be used.

```
Plus @@ (#^2& /@ myList)
```

Here is a comparison of various groupings.

```
{Plus @@ #^2& /@ myList, (Plus @@ #^2)& /@ myList, Plus @@ (#^2& /@ myList)
```

`Map` and `Apply` have the same precedences. The rightmost elements are grouped together.

```
Hold[a @@ b /@ c] // FullForm
```

```
Hold[a /@ b @@ c] // FullForm
```

Also, if only Apply and Map are nested they group from the right.

```
Hold[a @@ b @@ c @@ d] // FullForm
```

```
Hold[a /@ b /@ c /@ d] // FullForm
```

Once in a while, we need to perform some operation on the individual elements of a list, but the operation may not give a (wanted) result for some elements, in which case, that element is to be removed from the list. In such a situation, we could make the result of the operation Null, and then remove occurrences of Null using DeleteCases, or pick out elements other than Null using Select or Cases. This process can also be done directly by inserting Sequence[] at the corresponding places, although the following function does not immediately work.

```
Sequence[]&
```

Thus we take the following approach (based on the HoldAll attributes of Function).

```
(Sequence @@ {})&
```

It leads to the following program.

```
If[# > 0, #, Sequence @@ {}]& /@ {0, -1, 1, -2, 2, -3, 3, -4, 4, -5, 5}
```

Giving Function the HoldAllComplete attribute results in the following behavior.

```
SetAttributes[Function, HoldAllComplete];
If[# > 0, #, Sequence[]]& /@ {0, -1, 1, -2, 2, -3, 3, -4, 4, -5, 5}
```

Giving If the HoldAllComplete attribute also does not give the intended result.

```
SetAttributes[If, HoldAllComplete];
If[# > 0, #, Sequence[]]& /@ {0, -1, 1, -2, 2, -3, 3, -4, 4, -5, 5}
```

We remove the attribute HoldAllComplete from Function and from If.

```
ClearAttributes[Function, HoldAllComplete];
ClearAttributes[If, HoldAllComplete];
```

In graphics applications, we are often given a list of elements with the following structure.

```
Clear[a, b, c, d, f]
graphicsList = Table[
  ToExpression["f[g" <> ToString[k] <> "[i" <> ToString[k] <>
               ", j" <> ToString[k] <> "]]"], {k, 12}]
```

```
Clear[f, g, i, j, k]
graphicsList = Table[f[Subscript[g, k][
                          Subscript[i, k], Subscript[j, k]]], {k, 12}]
```

Now, we apply a function (e.g., κ) to all the g_i. This process can be done with Map.

```
Map[κ, graphicsList, {2}]
```

Here are all the expressions from level {2}.

```
Level[graphicsList, {2}]
```

Mapping a function to the level {0} changes the head.

```
Clear[f, a, b, c, d];
Map[f, {{a, b}, {c, d}}, {0}]
```

Mapping a function to an atom does return the atom. (Map by default maps to level {1}. The level {1} of an atom is the empty list { }. Mapping the function to the empty list results in the empty list. So the atom returns.)

```
Map[Sin, 1]
```

Mapping at level {0} means function application.

```
Map[Sin, 1, {0}]
```

To work on all levels simultaneously, we can use `MapAll`.

`MapAll[`*function*`,` *list*`]`
or
function `//@` *list*

applies the function *function* to all elements at all levels of the list *list*. The head of *list* need not be `List`.

In the next example, f is applied to every element at every level.

```
f //@ {{1, 2, 3}, {4, 5, 6}, {2}, {{w}}, j}
```

The application of `f` occurred a total of 15 times.

```
Length[Position[%, f[_]]]
```

The two commands `Map` and `MapAll` have the option `Heads`, as do most commands involving `Level` specifications.

```
{Options[Map], Options[MapAll]}
```

Here is a comparison between `Heads -> False` (default) and `Heads -> True`.

```
MapAll[nis, {Sin[Sin[y]], Sin[Sin[y]]}]

MapAll[nis, {Sin[Sin[y]], Sin[Sin[y]]}, Heads -> True]
```

Using the option setting `Heads -> True`, we can also map on heads.

```
Nest[MapAll[#, #, Heads -> False]&, ¡[1], 2]

Nest[MapAll[#, #, Heads -> True]&, ¡[1], 2]
```

The following somewhat unusual example does a good job of illustrating the operation of `MapAll`. The program exchanges the heads and arguments.

```
Clear[x, g, a, b, c, Ꭿ, ϒ];

MapAll[# /. {p1___[p2___] :> (* head[args] ⟶ args[head] *) p2[p1]}&,
       Exp[Sin[x^2 + g[a, b, c]^3] + Ꭿ[ϒ]^2 + 3]]
```

To understand this result better, we look at the following simpler case.

```
MapAll[# /. {p1__[p2__] :> p2[p1]}&, Exp[a + b]]
```

Here is another example with `MapAll`. It shows nicely the order of evaluations, as already discussed in Chapter 4.

```
Clear["f*"]

(Print["Now evaluating  ", #, "."]; #)& //@
          f3[f21[f211, f212], f21[f221, f222]]
```

Here, we have `Heads -> True`. `f3` and `f21` are now also printed.

```
MapAll[(Print["Now evaluating  ", #, "."]; #)&,
       f3[f21[f211, f212], f21[f221, f222]], Heads -> True]
```

If a function is to be applied only to particular elements rather than all elements, we use `MapAt`.

> `MapAt[`*function*`, `*list*`, `*positionSpecification*`]`
>
> applies *function* to the elements in *list* in the positions specified by *positionSpecification*. The head of *list* need not be `List`.

Here is a matrix.

```
mat = Table[{i, j}, {i, 4}, {j, 4}]
```

Here are some selected elements enclosed with `usl`.

```
MapAt[usl, mat, {{1, 2}, {4, 4}, {3, 3}}]
```

Let us discuss how to manipulate parts of expressions. One possibility is the `MapAt` command. *Mathematica* also allows the direct manipulation of a part of an expression *expr* in the form *expr*`[[`*part*`]] = `*newValue*. This method of changing parts is very fast. Here is an example with a list of 10000 elements.

```
Remove[testList, testList1];
```

```
testList = Range[10000];
```

This input changes the first 1000 elements of `testList`.

```
Do[testList[[i]] = i + 1, {i, 1000}] // Timing
```

The corresponding `MapAt` version is much slower.

```
testList1 = Range[10000];
testList1 = MapAt[(# + 1)&, testList1, List /@ Range[1000]]; // Timing
```

Here is another slow version.

```
testList2 = Range[10000];
Do[testList2 = ReplacePart[testList2, i + 1, i], {i, 1000}]; // Timing
```

Calling `ReplacePart` with four arguments is still slower.

```
testList3 = Range[10000];
testList3 = ReplacePart[testList3, Range[2, 1001],
            List /@ Range[1000], List /@ Range[1000]]; // Timing
```

All four lists `testList`*i* are identical.

```
testList === testList1 === testList2 === testList3
```

The construction *expr*`[[`*part*`]] = `*newValue* is so fast because it does not evaluate the whole expression *expr* after the replacement. We can see this behavior by replacing `smallList`, the first element in the following list, by a `Sequence`.

```
smallList = {1, 2}
```

```
smallList[[1]] = Sequence[3, 4]
```

The `Sequence` command did not disappear.

```
??smallList
```

If we evaluate `smallList`, `Sequence` disappears.

```
smallList
```

But in the list of downvalues, it is still there.

```
??smallList
```

Using a list inside the right-hand side allows us to set more than one element at a time.

```
a = {1, 2, 3, 4, 5, 6}

a[[{2, 4, 6}]] = {1, 3, 5}

a
```

Another function in the `Map` family is `MapIndexed`.

`MapIndexed[`*function*`,` *expression*`,` *levelSpecifications*`]`

applies the function *function* to the elements of *expression* at level *levelSpecifications*, where *function* gives the description of the position of the elements as its second argument. The usual level specifications are used for *levelSpecifications*. The head of *expression* need not be `List`.

In the following example, we use a function that evaluates to nothing but itself to improve readability.

```
mytab = Table[𝕩[i, j, k], {i, 2}, {j, 2}, {k, 2}]
```

Here, we apply 𝕠 to each 𝕩 along with the position specification of each 𝕩.

```
MapIndexed[𝕠, mytab, {3}]
```

`MapIndexed` also carries the option `Heads`. We now give an example with a somewhat more complicated result. Note that when the function is applied to heads, zeros appear in the second argument of each 𝕠.

```
MapIndexed[𝕠, mytab, {3}, Heads -> True]
```

This input maps the function 𝕠 to every possible position.

```
MapIndexed[𝕠, mytab, {0, Infinity}, Heads -> True]
```

`MapIndexed` is often a very useful function to color graphics objects according to order. For example, consider the map $\{q, p\} \to \{q', p'\}$ (a, b fixed)

$$\{q', p'\} = \begin{cases} \left\{\dfrac{q}{a}, p\,a\right\} & 0 \le q \le a \\ \{1 - p, q\} & a < q < b \\ \left\{\dfrac{q-b}{1-b}, p(1-b) + b\right\} & b \le q \le 1. \end{cases}$$

and apply it iteratively to the point {0.16, 0.5}.

```
With[{a = 0.33, b = 0.66},
points =
NestList[Which[0 <= #[[1]] <= a, {#[[1]]/a, #[[2]] a},
               a <  #[[1]] <  b, {1 - #[[2]], #[[1]]},
               b <= #[[1]] <= 1, {(#[[1]] - b)/(1 - b),
                                  #[[2]](1 - b) + b}]&,
         {0.16, 0.5}, 50000]];
```

We can color the points with hues between red and blue, allowing us to see the order in which they were created. (The details of graphics displays are discussed in the next chapter.)

```
color[x_] := Hue[0.78 x[[1]]/20001];

Show[Graphics[{PointSize[0.005],
                        (* color points according to position *)
                        MapIndexed[{color[#2], #1}&, Point /@ points]}],
     AspectRatio -> Automatic, PlotRange -> All,
     Frame -> True, FrameTicks -> None]
```

We see an ordered arrangement of regions of completely different structures in this map; a detailed explanation is not possible here; see [181✱].

WriteRecursive

We now use the above-described possibilities of manipulating lists to program a function `WriteRecursive` that shortens a long *Mathematica* expression by recursively replacing all subexpressions appearing more than once with temporary symbols.

First, we look at all elementary expressions in the expressions to be manipulated, and replace all elementary "types" by new expressions. Then, we look at all expressions that have depth two, and again replace these with new expressions. We repeat this process until the entire expression consists only of one temporary symbol. All this is done in the program `WriteRecursive`.

We construct the replacement rule *temporarySymbol* ⇒ *expression* in the form `RuleDelayed[`*temporarySymbol*`,` *expression*`]`. This form has two advantages. First, the `HoldRest` attribute of `RuleDelayed` prevents an immediate calculation, and second, the result can later be easily expanded using `ReplaceRepeated`. The temporary symbols used have the form *userDefinedNameIncreasingInteger*. The values of variables with the same name may be erased in the process. It is possible to get the behavior of `WriteRecursive` in a somewhat more elegant way, but here we are focusing on other aspects. We also barely test the variables for their type, and `WriteRecursive` is not fully developed in other ways (see below). To make `WriteRecursive` a robust program would still require some work. (Because this is the first larger program discussed in detail in this book, we do not want to overdo it.)

We use one of the four zeros of the polynomial $12\,x^4 + 78\,x^3 + 56\,x^2 + 89\,x + 44 = 0$ of degree 4 as our test expression. This is a very long expression; we return to the question of how to compute it using `Solve` in detail in Chapter 1 of the Symbolics volume [303✱].

```
(largeTestExpression = #[[1, 2]]& /@
      Solve[12 x^4 + 78 x^3 + 56 x^2 + 89 x + 44 == 0, x]) // InputForm

LeafCount[largeTestExpression]
```

Here is our program for `WriteRecursive`. Because it is the first larger program presented, we give extensive comments in the code.

```
WriteRecursive[expression_(* expression to be simplified *),
               recv_Symbol(* auxiliary variable for the
             recursive definition *)] :=
 Module[(* definition of the local variables *)
        {expressionNew, index, depth, low, replacementList,
        invertedReplacementList, temp, invertedTemp},
 (* clearing global variables might be dangerous;
  a "production code" should be refined here *)
 Clear[Evaluate[StringJoin[ToString[recv] <> "*"]]];
 expressionNew = expression;
 (* variables on the left in patterns cannot be given values
  temporarily on the right;
```

```
   so, we make a copy of the original expression *)
  (* analyze the depth of the expression,
   assign index variables, and
   define a "working" expression *)
  depth = Depth[expressionNew];
  index = 0;
  (* check the current status of the messages
   General::spell1 and General::spell1 *)
  generalSpellWasOn  = If[Head[General::spell ] === String, True, False];
  generalSpell1WasOn = If[Head[General::spell1] === String, True, False];
  (* turn off the warning about similar-named variables,
   because in using the replacement, a lot of similar-named
   variables of the form recvnumber will appear *)
  Off[General::spell1]; Off[General::spell];
  (* find the leaves *)
  low = Union[Level[expression, {-1}]];
  (* replace equal leaves by the same symbol,
   and increase index *)
  replacementList = ToExpression[
      ToString[recv] <> ToString[index = index + 1]] :> #& /@ low;
  (* "invert" the replacement list obtained
    (i.e., form recvnumber -> subexpression *)
  invertedReplacementList = Reverse /@ replacementList;
  (* insert the temporary variables just created
   into the initial expression *)
  expressionNew = expressionNew //. invertedReplacementList;
  (* start a loop that continues until all levels
   of expression have been run through *)
 Do[(* find all subexpressions of depth 2 *)
    low = Union[Level[expressionNew, {-2}]];
    (* replace the same expressions at the depth 2 by
   the same symbol *)
    temp = ToExpression[ToString[recv] <> ToString[
                          index = index + 1]] :> #& /@ low;
    (* "invert" the replacement list obtained.
   The command Flatten is discussed in the next section.
   It removes inner pairs of braces {} *)
    replacementList = Flatten[AppendTo[replacementList, temp]];
    (* insert the temporary variables in the initial expression *)
    invertedTemp = Reverse /@ temp;
    expressionNew = expressionNew //. invertedTemp, {depth}];
    (* turn on the warning again in case
   they were turned on before *)
    If[generalSpellWasOn, On[General::spell]];
    If[generalSpell1WasOn, On[General::spell1]];
    (* print out the resulting replacement list *)
   replacementList]
```

Here is the result for the example above with four zeros. It has clearly been shortened.

```
WriteRecursive[largeTestExpression, temp]

LeafCount[%]
```

To check this result, we insert everything and compare it with the initial expression.

```
(First[Last[%%]] //. %%) == largeTestExpression
```

Here is another example.

```
Nest[1 + 1/(1 + Sin[Sqrt[#]])&, x, 4]

WriteRecursive[%, ϒ]

%[[-1, 1]] //. %
```

As mentioned, our `WriteRecursive` is still not completely refined. Because we always pick out the level `{-2}`, all sums and products of atoms are pulled out at once. This also happens when they contain many common terms.

```
WriteRecursive[𝔾[x1 + x2 + x3 + 4, x1 + x2 + x3 + 5], λ]
```

For some details about rewriting a given expression in terms of common subexpressions, see [58✱], and [105✱].

Also, if the expression to be written recursively has `Hold`-like parts, they will not be correctly handled in the implementation above.

```
Hold[1 + 1]

WriteRecursive[Hold[1 + 1], tr]

tr4 //. %
```

With some extra work, we could take account of such special cases using methods similar to those in Section 6.6. A possible purpose of `WriteRecursive` is to shorten large arithmetic expressions (and to speed up their numerical evaluation), which arise, for example, in the exact solution of large equations and systems of equations; we will make use of it later again. A very elaborate function that rewrites expressions in such a form so that its numerical evaluation is more efficient can be found in the context `` Experimental` ``. In the above implementation, we did not care about the runtime of `WriteRecursive` as a function of the size of its first argument. The function `OptimizeExpression` does care about this complexity.

```
??Experimental`OptimizeExpression
```

Here is our example from above, rewritten.

```
Experimental`OptimizeExpression[largeTestExpression]
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

■ 6.3.4 Arithmetical Properties of Lists

In this short subsection, we will discuss a few built-in functions that easily allow determining some arithmetic properties of a single list. We already discussed the function `Length`, which gives the length of a list. The functions `Min` and `Max` allow obtaining the smallest and largest element of a list. The average value of a list can be obtained through the function `Mean`.

`Mean[`*list*`]`

gives the average value of the list *list*.

The sum of all elements is obtained through the function `Total`.

`Total[`*list*`]`

gives the sum of the elements of the list *list*.

The square root of the sum of the absolute values of a list can be obtained through `Norm`.

`Norm[`*list*`]`
gives the (2) norm of the list *list*.

In addition to mean, the most important statistical property of a list is its variance $1/(n-1)\sum_{k=1}^{n}\left(l_k-\bar{l}\right)^2$ where $\bar{l}=\sum_{k=1}^{n} l_k$ is the mean of a list with elements $l_1, l_2, \ldots, l_n$.

`Variance[`*list*`]`
gives the variance of the list *list*.

Here is a series of simple example for these three functions. We use a list with symbolic elements. This allows to easily recognize the functional form of the results.

```
L = {a, b, c, d};
```

```
Total[L]
```

```
Norm[L]
```

```
Mean[L]
```

```
Variance[L]
```

Here is the mean of a list with five elements. But three of the list elements disappear because they have the head `Sequence`.

```
Mean[Unevaluated[{a, Sequence[], Sequence[], Sequence[], e}]]
```

For approximative numeric arguments, these quantities collapse to numbers.

```
cosList = Table[N[Cos[k]], {k, 10^6}];
```

```
#[cosList]& /@ {Mean, Total, Variance}
```

Sometimes one needs to "approximately partition" a list.

`Quantile[`*list*`, `*quantile*`]`
gives the quantile of the list *list*.

The quantile `Quantile[`*l*`, `*q*`]` is defined through `Sort[`*l*`, Less][[Ceiling[`*q* `Length[`*l*`]]]]`. The quantile *q* must be in the range [0, 1].

Here is a list of length 100 and the quantile 0.1 and 0.4 are determined.

```
Quantile[Range[100], {0.1, 0.4}]
```

Because of the use of `Ceiling` in the definition of `Quantile`, the following gives the result 51.

```
Quantile[Range[100], 1/2 + 10^-100]
```

For lists with symbolic elements, the quantile cannot be determined.

```
Quantile[{a, b, c, d, e}, 1/2]
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

6.4 Operations with Several Lists or with Nested Lists

■ 6.4.1 Simple Operations

If the sizes of two or more lists are equal, the elements of these lists can be added, subtracted, multiplied, divided, and raised to powers elementwise. We now look at this elementwise application of +, -, *, /, and ^ for two example matrices.

```
mat1 = {{a11, a12}, {a21, a22}};
mat2 = {{b11, b12}, {b21, b22}};
```

Here is their sum. The elements are added because of the `Listable` attribute of `Plus`.

```
mat1 + mat2
```

This is their product. The elements are multiplied because the `Listable` attribute of `Times`. This gives the Hadamard product [192✱].

```
mat1 mat2
```

Here is their quotient.

```
mat1/mat2
```

We raise one matrix to the power of the other.

```
mat1^mat2
```

The transpose of a matrix can be found with `Transpose`.

`Transpose[`*list*`, {`i_1`,`i_2`, …, `i_n`}]`

“transposes” the levels of the nested list *list* as follows: $level_1 \longrightarrow level_{i_1}$, $level_2 \longrightarrow level_{i_2}, \ldots$, $level_n \longrightarrow level_{i_n}$. Here, *list* must be a rectangular matrix. The head of *list* need not be `List`. If the list `{`i_1`, `i_2`, …, `i_n`}` does not appear, only the first two levels are exchanged; that is, in this case the second argument is `{2, 1, 3, … }`.

Thus, if **M** is a matrix with elements m_{ij}, the elements of the transposed matrix $\mathbf{M}^{\mathrm{T}}$ are $\mathbf{M}_{ji}$.

```
(M = {{a11, a12}, {a21, a22}}) // TableForm
```

```
Transpose[M] // TableForm
```

Here is an example with three levels.

```
threeMat = Table[a[i, j, k], {i, 3}, {j, 3}, {k, 2}]
```

We want to look at all possible applications of `Transpose` to `threeMat` with all possible second arguments. To automatically generate all possible cases, we need one other list command.

`Permutations[`*list*`]`

gives a list of all possible permutations of the elements in the list *list*. The head of *list* need not be `List`.

For example, here are all permutations of the list {1, 2, 3}.

```
Permutations[{1, 2, 3}]
```

Again, the head need not be `List`.

```
Permutations[F[a, b, c]]
```

Now, we can transpose `threeMat` in "different ways".

```
(CellPrint[Cell[TextData[{"∘ ",
                StyleBox["Transpose[threeMat, " <>
                          ToString[{##}] <> "]", "MR"],
                          " yields the following:"}], "PrintText"]];
 Print[TableForm[Transpose[threeMat, #]]])& /@ Permutations[{1, 2, 3}];
```

Let us also look at what happens when two or three of the permutation indices coincide. Here is a list of all permutations.

```
perms = Union[Flatten[Permutations /@
          Flatten[Outer[List, #, #, #]&[{1, 2, 3}], 2], 1]]
```

These are the second arguments of `Transpose`, which are treated without generating a message.

```
DeleteCases[Check[Transpose[threeMat, #]; #, {}]& /@ perms, {}]
```

We need two or more identical indices.

```
Select[%, Length[Union[#]] < 3&]
```

These are the matrices after the application of `Transpose`.

```
Transpose[threeMat, #]& /@ %
```

When identical integers appear in the second argument list of `Transpose`, we have the following behavior.

```
Transpose[Table[A[i, j, k], {i, 3}, {j, 3}, {k, 3}], {1, 1, 1}]

Transpose[Table[A[i, j, k], {i, 3}, {j, 3}, {k, 3}], {1, 1, 1}]
```

We see that the corresponding diagonal elements were picked out.

In addition to the exchange of levels of lists, it is also possible to remove inner brackets. To do that, the `Flatten` command is useful. (Of course, one could use `Apply[List,` *expressions*`,` *levels*`]`, but this is not very convenient.)

`Flatten[`*list*`,` *n*`]`

removes the inner brackets in the top *n* levels of the (maybe nested) list *list*. If the second argument is not present, all inner brackets are removed. The head of *list* need not be `List`.

Here is a fourfold nested list.

```
ma = Table[i + j + k + l, {i, 5}, {j, 3}, {k, 4}, {l, 2}]
```

Now, we remove the pairs of brackets, starting at the top.

```
Flatten[ma, 1]

Flatten[ma, 2]

Flatten[ma, 3]
```

To remove all lists from all sublevels, the second argument need not be given explicitly.

```
am = Flatten[ma]
```

Other heads, here those with `f`, can also be removed.

```
Flatten[f[f[a, b], f[f[a, b], f[a, b]]]]
```

This removal is carried out if the heads are continuously present, starting at the top level.

```
Flatten[f[f[{{f[f[ξ]]}}]]]
```

Using `MapAll`, we can flatten all nested identical heads. But `Flatten[`*atom*`]` will not evaluate to *atom*.

```
MapAll[Flatten, f[f[{{f[f[ξ]]}}]]]
```

Now that we have introduced the command `Flatten`, we return for a short time to the command `AppendTo`. For recursive construction of long lists, `AppendTo` is not appropriate because it is very slow. Suppose we want to construct a list containing 5000 elements. In the following two approaches, we add one element at a time.

```
Timing[li = {}; Do[AppendTo[li, i], {i, 5000}]; Length[li]]

Timing[li = {}; Do[li = Append[li, i], {i, 5000}]; Length[li]]
```

We now form a new list consisting of the old list together with the element to be appended, and then remove the inner brackets around the old list. The following approach is even slower.

```
Timing[li = {}; Do[li = Flatten[{li, i}], {i, 5000}]; Length[li]]
```

A much more efficient approach is to nest the lists 50000 times (ten times more elements than in the last example), and then remove all inner brackets at one time. (Note that the difference in the computed time is again roughly an order of magnitude.)

```
Timing[li = {}; Do[li = {li, i}, {i, 50000}]; Length[Flatten[li]]]
```

Another fast method to construct lists of a priori unknown length is the use of `Sow` and `Reap`. The next input builds again a list of length 50000. The time needed to this is approximately equal to the one from the last example.

```
Timing[Length[Reap[Do[Sow[i], {i, 50000}]][[2, 1]]]]
```

Here, the timings of the same approaches are graphically shown for variable list size (for a detailed comparison of these approaches, see [326✱]).

```
(* common options for the next three graphics *)
opts[label_] := Sequence[PlotRange -> All, DisplayFunction -> Identity,
  AxesLabel -> {"list length", "t/s"},
  PlotLabel -> StyleForm[label, FontFamily -> "Courier", FontSize -> 10]]

With[{(* different sizes for good graphics and minimal timings *)
      n1 = 1500, n2 = 200, n3 = 2000},
Show[GraphicsArray[{  (* Append *)
ListPlot[Array[Timing[Nest[Append[#, 1]&, {}, #]][[1, 1]]&, n1],
         Evaluate[opts["Append"]],
         Ticks -> {{500, 1000, 1500}, Automatic}],
Module[{l = {}},        (* AppendTo *)
ListPlot[Array[Timing[Nest[AppendTo[l, 1]&, 1, #]][[1, 1]]&, n2],
         Evaluate[opts["AppendTo"]],
         Ticks -> {{100, 200}, Automatic}]],
                        (* Flatten *)
ListPlot[Array[Timing[Flatten[Nest[{#, 1}&, 1, #]]][[1, 1]]&, n3],
         Evaluate[opts["Flatten"]],
         Ticks -> {{1000, 2000}, Automatic}]}]]]
```

Here is an application for the just-discussed method of collecting data: In the following calculation, we put all functions that have a real argument in `realBag`. We achieve this result by putting the function in the bag `realBag` as a side effect of testing if the rule is applicable (the test `# =!= real&` serves to avoid recursion).

```
Unprotect[Real];

realBag = real[];

Real /: f_?(# =!= real&)[___, x_Real, ___] :=
             (Null /; (realBag = real[realBag, x]; False))
```

Now, we calculate a numerical value of a hypergeometric function (see Chapter 3 of the Symbolics volume [303✱]).

```
HypergeometricPFQ[{1.3, 4.5, 2.3}, {2.1, 2.3, 2}, 2.34]
```

We then look at `realBag` to see what has been collected in it.

```
{Depth[realBag], Length[realBag],
 Depth[Flatten[realBag]], Length[Flatten[realBag]]}
```

Here are the smallest and largest numbers encountered in calculating ${}_3F_3(1.3, 4.5, 2.3; 2.1, 2.3, 2; 2.34)$.

```
{Min[#], Max[#]}&[Abs[Cases[realBag, _, {-1}]]]
```

Here, the same is done for the head `Integer` and for heads (symbols). This time, we also collect the function names.

```
Unprotect[Integer];

integerBag = integer[];
headBag = head[];

Integer /: f_?(# =!= integer && # =!= F&)[___, x_Integer, ___] :=
             (Null /; (integerBag = integer[integerBag, x];
                       headBag = F[headBag, f]; False))
```

Again, we evaluate a generalized hypergeometric function.

```
HypergeometricPFQ[{9, 6, -5}, {4, 6, 7}, 12]

{Depth[integerBag], Length[integerBag],
 Depth[Flatten[integerBag]], Length[Flatten[integerBag]]}
```

These heads (functions) were used in the calculation.

```
Cases[Union[Flatten[headBag]], _Symbol]
```

The method above of assigning rules that are never applicable is a useful additional tool for debugging to find out with which arguments, how often, and so on various functions are called. We will make use of this technique in later chapters.

Now, we destroy the definition above because it slows down all calculations considerably.

```
Unprotect[Real];
UpValues[Real] = {};
Protect[Real];

Unprotect[Integer];
UpValues[Integer] = {};
Protect[Integer];
```

After this little excursion, let us come back to flattening lists. The `FlattenAt` command is somewhat more specific than `Flatten`.

`FlattenAt[`*list*`, `*positionsList*`]`

removes the inner brackets around the elements in list at the positions defined by *positionsList*. The head of *list* need not be `List`.

```
MA1 = {1, {2, {3, 4}}}
```

Here again all inner brackets vanish.

```
Flatten[MA1]
```

Now, we remove only the inner brackets around 3 and 4.

```
FlattenAt[MA1, {2, 2}]
```

The converse of `Flatten` can be obtained with `Partition`.

`Partition[`*list*`, {`i_1`, `i_2`, ..., `i_n`}, {`$offset_1$`, `$offset_2$`, ..., `$offset_n$`}]`

partitions *list* into i_k parts with offsets of $offset_k$ elements in the *k*th step. Here, i_k, $offset_k > 0$ must be integers. The head of *list* need not be `List`.

```
mα = Table[i + j + k + l, {i, 5}, {j, 3}, {k, 4}, {l, 2}]
```

Because we created mα using `Table[i + j + k + l, {i, 5}, {j, 3}, {k, 4}, {l, 2}]`, we can get back the original matrix as follows (the last level arises automatically in the partition).

```
mα

Partition[Partition[Partition[αm, 2], 4], 3]
```

Here is a somewhat more elegant solution.

```
Fold[Partition, αm, {2, 4, 3}]
```

An explicit comparison verifies the results.

```
mα == % == %%
```

We now consider lists with lengths 1 through 6 with different offsets in `Partition`. We display a condensed version of the result. We have six possible resulting lists for the 42 possibilities. The first elements of each list are the values of the {*i*, *j*}-pair that generate the second element.

```
{First /@ #, #[[1, 2]]}& /@
Split[Sort[Flatten[
Table[{{i, j}, Partition[{1, 2, 3, 4, 5}, i, j]},
      {i, 6}, {j, 7}], 1], #1[[2]] === #2[[2]]&], #1[[2]] === #2[[2]]&]
```

The following input produces a more readable, although much larger, output.

```
Do[CellPrint[Cell[TextData[{
  StyleBox["◦ Partition[{1, 2, 3, 4, 5}, " <>
            ToString[i] <>", " <> ToString[j] <>"]", "MR"],
             " yields  ", StyleBox[ToString[(* the partition *)
                 Partition[{1, 2, 3, 4, 5}, i, j]], "MR"], "." }],
                   "PrintText"]], {i, 6}, {j, 7}]
```

Note the case *offset* = *j* > *i* in the last example. Here is another example to show that, in this case, some elements in the list do not appear in the result at all.

```
Partition[{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13}, 3, 4]
```

If we consider lists as sets of elements, it is possible to define the following operations: forming the union, intersection, and the complement. First, we consider the conjunction of elements (which is not too meaningful from a set theory standpoint because equal element are kept, but it is very useful in programming).

`Join[`$list_1$`, `$list_2$`, …, `$list_n$`]`

gives a list of all elements in the given lists $list_i$. The elements appear in the same order as in the lists $list_i$. The head of $list_i$ need not be `List`.

Here is a simple example that joins two lists.

```
Join[{1, 2}, {3, {4}}]
```

Here, we apply `Join` to two objects with head `hh`.

```
Join[ℍ[5, 6], ℍ[7, 8, 5, 6]]
```

Although the arguments do not necessarily have the head `List`, they must all have the same head to be joined.

```
Join[{1, 2}, list[3, 4]]
```

The next command forms the set theoretic union. (The above-mentioned application of `Union` is just that, in the sense that in a set, every element can occur only once, which is a special case of the following.)

`Union[`$list_1$`, `$list_2$`, …, `$list_n$`]`

gives a list of all elements of all given lists $list_i$, sorted according to the canonical order. Elements that appear more than once in the $list_i$ are included just once in the result. The head of $list_i$ need not be `List`.

To get the intersection, we use `Intersection`.

`Intersection[`$list_1$`, `$list_2$`, …, `$list_n$`]`

gives a list of all elements that appear at least once in each of the lists $list_i$. Elements appearing more than once are included just once in the result. The head of $list_i$ need not be `List`.

Finally, we look at forming the complement.

`Complement[`*relativeTo*`, `$list_1$`, `$list_2$`, …, `$list_n$`]`

gives a list of all elements appearing in *relativeTo*, but not in any of the $list_i$ ($i = 1, \ldots, n$). The head of $list_i$ need not be `List`.

Here are three self-explanatory examples.

```
Complement[{1, 2, 3, 4, 5, 6, 7, 8, 9}, {1}, {2}, {3}, {4}]

Intersection[{1, 1, 2, 3, 4, 5, 6, 7, 8, 9}, {1, 1},
             {1, 1, 2}, {1, 1, 3}, {1, 1, 4}]

Union[{1, 2, 3, 4, 5, 6, 7, 8, 9}, {1}, {2}, {3}, {4}]
```

The three commands `Union`, `Complement`, and `Intersection` possess one option (just like `FixedPoint` and `FixedPointList` discussed in Chapter 3).

```
Options /@ {Union, Complement, Intersection}
```

`SameTest`

is an option for the commands `Union`, `Complement`, and `Intersection`. It defines when two elements are to be treated as identical.

Default:

`Automatic`

Admissible:

`Equal`, `SameQ`, or an arbitrary (pure) function of two variables

By using the default `SameTest` in `Union`, we get three elements in the following example.

```
Union[{1.0, 2.0}, 2{1, 1}]
```

But the following input gives two elements.

```
Union[{1.0, 2.0}, 2 {1, 1}, SameTest -> Equal]
```

We use a less restrictive test for comparison. It also returns a list with two elements.

```
Union[{1.0, 2.0}, 2 {1, 1}, SameTest -> (Abs[#1 - #2] < 10^-15&)]
```

At this point, let us make a remark about a potential pitfall when working with `Union`. `Union[`*list*`]` works by first sorting *list* and then comparing adjacent elements, which has the advantage that it can be done fast, having a complexity $O(l \log(l))$, where l is the length of *list*. The disadvantage of this presorting is that elements that might be considered the same are not sorted adjacent to each other and as a result are kept. Here is an example: `numberList` is a list of 162 numbers, all are closely centered around $1. + 1.\, i$ and $1. - 1.\, i$.

```
numberList =
Flatten[{(* near 1 + I *)
          Table[1.0 +  1.0 I + (j + I k) $MachineEpsilon,
                {j, -2, 2, 1/2}, {k, -2, 2, 1/2}],
           (* near 1 - I *)
          Table[1.0 - 1.0 I + (j + I k) $MachineEpsilon,
                {j, -2, 2, 1/2}, {k, -2, 2, 1/2}]}];
```

Just applying `Union` to this list leaves many elements in this list.

```
(unionedNumberList = Union[numberList]) // Length
```

The minimal distance between two numbers in the list `unionedNumberList` is smaller than `$MachineEpsilon`.

```
Min[Table[Abs[unionedNumberList[[i]] - unionedNumberList[[j]]],
          {i, Length[unionedNumberList]},
          {j, i + 1, Length[unionedNumberList]}]]/$MachineEpsilon
```

Using an explicit setting for `SameTest` forces *Mathematica* to use a slower algorithm, which has quadratic complexity. Now, only two elements remain after applying `Union`.

```
Union[numberList, SameTest -> (#1 == #2&)] // Length
```

```
Union[numberList,
      SameTest -> (Abs[#1 - #2] < 6 $MachineEpsilon&)] // Length
```

The use of *any* `SameTest` forces the use of an algorithm with quadratic complexity versus an $O(n \ln(n))$ complexity. The following two inputs clearly show this change in complexity.

```
Table[list = Range[10^k];
      Timing[Union[list]][[1]], {k, 3, 6}]
```

```
Table[list = Range[10^k];
      Timing[Union[list, SameTest -> Equal]][[1]], {k, 1, 4 (* ! *)}]
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

■ 6.4.2 List of All System Commands

Now that we are able to manipulate lists, we take another look at "meta-*Mathematica* things", similar to issues that were discussed in Chapter 4. The following considerations and examples are not of practical use, but they are given as examples of the use of larger lists and how to manipulate them. Here we will make heavy use of `Map` and `Apply` (this means the input forms `/@` and `@@` will appear frequently in this subsection). We want to investigate some properties of all built-in commands. An essential step is the following construction, which converts strings into the corresponding nonevaluated expressions. (We need this because `Names["*"]` gives a list with the names of all built-in commands in the form of strings.)

```
x = 1; y = 2; z = 3;
Apply[Unevaluated, #]& /@ (ToHeldExpression /@ {"x", "y", "z"})
```

Another possibility to achieve the same output is the use of `ToExpression[`*expr*`, InputForm, Unevaluated]`.

```
x = 1; y = 2; z = 3;
ToExpression[#, InputForm, Unevaluated]& /@ {"x", "y", "z"}
```

If we apply a *Mathematica* command to this, the argument of `Unevaluated` is not immediately evaluated. We did not use this construction in this form in Chapter 4. It was not needed for the commands dealt with there. Here is an excellent example of the operation of `Unevaluated`.

```
Head[Unevaluated[Print["AmIPrintedNow?"]]]
```

To be sure that we get only the system commands, we could restart *Mathematica* here and begin again with `Names["*"]` (this gives us the list of all built-in *Mathematica* commands), or we can use `Remove` to get rid of introduced symbols.

```
Remove[x, y, z]
Names["System`*"];
```

We call this list `allCommands`.

```
allCommands = DeleteCases[%, "$Epilog"];
```

This input gives the number of commands.

```
Length[allCommands]
```

If we simply convert the strings containing the names of the commands to commands, we get many nontrivial evaluations.

```
allCommandsEvaluated = ToExpression /@ allCommands;
```

The following commands were executed (now having a different value than they had in unevaluated form).

```
Complement[allCommands, ToString /@ allCommandsEvaluated]
```

```
Length[%]
```

Thus, we use the variant tested above.

```
allCommandsUnevaluated =
Apply[Unevaluated, #]& /@ (ToHeldExpression /@ allCommands);
```

Here are the first dozen elements of the resulting list.

```
Take[allCommandsUnevaluated, 12]
```

It has the desired structure. Now, for example, we can sort out all commands with options. To do this, we first generate a list `auxList` of all options of these commands selected and then measure its length. If it is not the empty list `{}`, the corresponding command has options, and otherwise it does not.

Here the list of all functions that have options.

```
auxList = Options /@ allCommandsUnevaluated;

commandsWithOptions = {};

Do[If[auxList[[i]] != {},
      AppendTo[commandsWithOptions, allCommandsUnevaluated[[i]]]],
   {i, Length[allCommandsUnevaluated]}];

 commandsWithOptions // Short[#, 12]&
```

(Another (and shorter) possible input to determine the functions with options would have been:
`Select[allCommandsUnevaluated, Options[#] =!= {}&]`.)
This is a total of about 200 functions.

```
Length[%]
```

`Notebook` is the command with the most options.

```
allCommands[[Position[#, Max[#]]&[Length /@ auxList][[1]]]]
```

It has about 200 options. (In most cases only a small fraction of these options are explicitly set to nondefault values.)

```
Length[Options[Notebook]]
```

The set of all possible options can be obtained as follows: First, remove all `{}` from `auxList` (using `Flatten`); then extract the first part, which is necessary because all options are in the form *option* `->` *actualDefault*; and finally, use `Union` to eliminate all options that appear more than once.

```
Union[Flatten[If[Length[#] > 0, First[#], {}]& /@
                   Flatten[auxList]]] // Short[#, 12]&
```

Here is the total number of options.

```
Length[%]
```

We turn now to the attributes by first counting the number of system commands having at least one attribute.

```
withAttributes = Select[allCommandsUnevaluated,
                          (Length[Attributes[#]] > 0)&];

Length[%]
```

Because these are nearly all built-in commands, we look instead at the set complementary to `withAttributes`. The following commands have no attributes.

```
ToString /@ Complement[allCommandsUnevaluated,
                         withAttributes] // Short[#, 12]&
```

Most commands have `Protected` as an attribute. The following commands have other nontrivial attributes.

```
withNotOnlyProtectedAttributes =
ToString /@ Select[withAttributes,
                Attributes[#] != {Protected}&] // Short[#, 12]&
```

This is the length of the list.

```
Length[%]
```

We find the attributes possessed by the built-in commands.

```
theAttributes = Union[Flatten[Attributes /@ allCommandsUnevaluated]]
```

It might happen that an existing attribute is not carried by any built-in command. Thus, we look at all usage messages to find those in which the word `"attribute"` appears. (The command `StringMatchQ` was discussed in Chapter 5; `StringMatchQ["`*string*`", "`*stringPattern*`"]` gives `True` if the string in the second argument appears in the first argument, and otherwise gives `False`.)

```
(* load all usage messages *)
Get[ToFileName[{$TopDirectory, "SystemFiles", "Kernel",
                "TextResources", $Language}, "Usage.m"]]

Off[StringMatchQ::string]; Off[StringMatchQ::strs];

theAttributesInTheUsageMessages =
ToString[#]& /@ Select[allCommandsUnevaluated,
                       StringMatchQ[MessageName[#, "usage"], "*attribute*"]]

On[StringMatchQ::string]; On[StringMatchQ::strs];

Complement[ToExpression /@ theAttributesInTheUsageMessages, theAttributes]
```

Indeed, such other attributes exist: `Stub` and `Temporary`. As expected, no built-in function has the `Temporary` attribute. And the `Stub` attribute has the following meaning.

```
??Stub
```

Here is the number of commands with the corresponding listed attributes.

```
Do[CellPrint[Cell[TextData[{"∘ The attribute ",
            StyleBox[ToString[theAttributes[[i]]], "MR"],
                        " is carried by " <>
             ToString[(* how many *) ℓ = Length[
      Select[allCommandsUnevaluated,
            Function[x, MemberQ[Attributes[x],
                                theAttributes[[i]]]]]]]] <>
                        (* singular or plural? *)
                        If[ℓ === 1, " command.", " commands."]}],
                         "PrintText"]], {i, Length[theAttributes]}]
```

Here is a list of all the symbols carrying the `Locked` attribute.

```
ToString /@ Select[allCommandsUnevaluated,
                   MemberQ[Attributes[#], Locked]&]

Length[%]
```

It is also interesting to see which commands carry the attributes `Flat`, `Orderless`, and `OneIdentity`.

```
ToString /@ Select[allCommandsUnevaluated,
                   MemberQ[Attributes[#], Flat]&]

ToString /@ Select[allCommandsUnevaluated,
                   MemberQ[Attributes[#], Orderless]&]

ToString /@ Select[allCommandsUnevaluated,
                   MemberQ[Attributes[#], OneIdentity]&]
```

These functions carry the three attributes `Orderless`, `Flat`, and `OneIdentity`.

```
ToString /@ Select[allCommandsUnevaluated,
                   (MemberQ[Attributes[#], Flat] &&
                    MemberQ[Attributes[#], Orderless] &&
                    MemberQ[Attributes[#], OneIdentity])&]
```

Options are given as rules, in most cases with `Rule`, but in some cases with `RuleDelayed`. Let us search for all default values of options that are realized with delayed rules.

```
Union[Flatten[Cases[#, _RuleDelayed]& /@
       Options /@ Apply[Unevaluated,
           ToHeldExpression /@ Names["*"], {1}]]]
```

Do any functions have more than one option and at the same time have the attribute `Listable`? This would mean that we could give a list of different options and obtain a list as the results. We first select the commands with the attribute `Listable` and then look at which ones have more than one option. (Here, it is safe to use `ToExpression` because none of these functions will evaluate to anything else.)

```
Select[Select[Names["*"], MemberQ[Attributes[#], Listable]&],
      Options[ToExpression[#]] =!= {}&]
```

`PrimeQ` is one such function. Calling `PrimeQ` with a list as its second argument results in an output with head `List`.

```
PrimeQ[17, {GaussianIntegers -> False, GaussianIntegers -> True}]
```

Next, we examine the names of the commands in more detail. For this procedure we need another string command.

`Characters[`*string*`]`
gives a list of the individual strings in the string *string*.

Here, the characters are a longer string.

```
Characters[" I consist of the following individual characters."]
```

Here is the list of the length of all command names.

```
lengthCommands = StringLength /@ allCommands;
```

The longest commands have 36 letters.

```
Max[%]
```

Here is the longest named function from the current contexts.

```
allCommands[[#[[1]]]]& /@ Position[lengthCommands, 36]
```

The shortest commands have one, two, or three letters.

```
allCommands[[#]]& /@ Flatten[Position[lengthCommands, 1]]

allCommands[[#]]& /@ Flatten[Position[lengthCommands, 2]]

allCommands[[#]]& /@ Flatten[Position[lengthCommands, 3]]
```

The next command that we need to count various things is `Count`.

`Count[`*list*`, `*toCount*`]`
gives the number of elements in the list *list* that have the pattern *toCount*. The head of *list* need not be `List`.

We can look at the distribution of the lengths of names in more detail. (For getting the count, we could also have used

StringLength.)

```
Table[{k, Count[lengthCommands, k]}, {k, 36}]
```

The average length of a *Mathematica* built-in symbol is about 12 characters.

```
Plus @@ lengthCommands/Length[lengthCommands] // N
```

Now, we can get the distribution of the starting letters. First, we "compute" a list auxList, in which the commands are replaced by a list of their letters. Then, we create a list of all starting letters. Finally, we simply count the number of commands starting with each letter using Union[First[#] & /@ auxList].

```
auxList = Characters[#]& /@ allCommands;

{#, Count[auxList, {#, ___}]}& /@ Union[First[#]& /@ auxList]
```

Mathematica commands start with capital letters, and each subword is also capitalized. To conclude, we also count how many capital letters are contained in the various *Mathematica* commands. (Here, we use the command UpperCaseQ, which we do not formally introduce because it is not used again; it gives True when its argument is a capital letter in the form of a string).

```
CellPrint[Cell[TextData["∘ There are " <> ToString[#[[2]]] <>
                                        " commands with "  <>
              ToString[#[[1]]] <> " capital letter" <>
          If[#[[1]] == 1, ".", "s."]], "PrintText"]]& /@
              (* count *)    Function[r, {#, Count[r, #]}& /@ Union[r]][
            Length[Select[#, UpperCaseQ]]& /@ Characters /@ allCommands];
```

Here are the *Mathematica* commands with six uppercase letters. They nearly form little sentences.

```
StringJoin /@
Select[Characters /@ Names["System`*"],
       Count[UpperCaseQ /@ #, True] === 6&]
```

We could go on and investigate the symbols from other contexts, such as Developer` and Experimental`.

We can do similar investigations, for instance, to determine whether any nontrivial (meaning of length > 1) palindromic names are built-in *Mathematica* commands.

```
Select[Names["System`*"], (# === StringReverse[#] && StringLength[#] > 1)&]
```

The chances for finding such a palindrome were small, because *Mathematica* built-in commands always start with a capital letter, but end typically with a lowercase letter. This output occurs if we do not differentiate between lowercase and uppercase letters.

```
Select[Map[ToLowerCase, Names["System`*"]],
       (# === StringReverse[#] && StringLength[#] > 1)&]
```

But do at least some names exist with letters that could be used to make another built-in name?

```
Function[ca,
 Function[lca,
          Do[Function[cai, If[# =!= {}, Print[
              (* the commands with equal letters *)
             Names["System`*"][[#]]& /@
                Flatten[Position[ca, #[[1]]]]]]&[
                     Select[Take[ca, {i + 1, lca}], (* the same? *)
          (# === cai)&]]][ca[[i]]], {i, 1, lca - 1}]][Length[ca]]][
                          Sort /@ Characters /@ Names["System`*"]]
```

Yes, fortunately, *Mathematica* has the Jacobi's elliptic functions (which we discuss in Chapter 3 of the Symbolics volume [303✱]) and some more from the front end area.

Next, we could investigate the average ratio of uppercase to lowercase letters in the *Mathematica* commands, and so on. We end here; the reader can continue this investigation if interested.

Next, we look at how many built-in commands have already some `DownValues`, before we give any definitions to them.

```
Off[General::readp];

DeleteCases[DownValues /@
    (Unevaluated @@ #& /@ (ToHeldExpression /@ Names["System`*"])),
           {} | $Failed] // Length
```

Here is the number of built-in commands that have `OwnValues`.

```
DeleteCases[OwnValues /@
    (Unevaluated @@ #& /@ (ToHeldExpression /@ Names["System`*"])),
           {} | $Failed] // Length
```

After studying the built-in names, we could go on to investigate *Mathematica* messages, then analyze the structure of programs, etc. Which message, for instance, is the longest one? To get information on all messages, we first have to remove the attribute `ReadProtected` from all commands.

```
Off[Protect::locked]; Unprotect["*"];
Off[Attributes::locked]; Off[ClearAttributes::sym];

(* make all accessible *)
ClearAttributes[#, ReadProtected]& /@
  ((Unevaluated @@ #)& /@ (ToHeldExpression /@ Names["System`*"]));
```

Here is the longest message.

```
Function[messageContent,
 messageContent[[#]]& /@ Flatten[Position[#, Max[
          Cases[#, _Integer]]]&[(* count number of strings *)
         StringLength /@ messageContent]]][Cases[#, _String]& @
             Flatten[Map[Last, (Messages /@
               ((Unevaluated @@ #)& /@ (ToHeldExpression /@
                                           Names["System`*"]))), {2}]]]
```

After having “finished” our investigations on built-in things (still a lot are possible), we go on with such “system investigations” by studying the internal dependency structure of packages. Which command from a package calls (potentially) which other command? Here is a possible implementation of this question. The program checks in the right-hand side of the definitions in which other commands appear and does this repeatedly, but not in infinite recursion. The arrow $\Longrightarrow$ in the result indicates dependencies. We take only definitions into account that are stored with `DownValues`. Extensions to include `OwnValues`, `UpValues`, `SubValues`, …, are straightforward to implement.

```
SetAttributes[symbolsUsed, HoldAll];

(* input is a string *)
symbolsUsed[expr_String] := symbolsUsed @@ {ToHeldExpression[expr]}

(* input is in the form Hold[symbol] *)
symbolsUsed[Hold[symbol_]] :=
Select[DeleteCases[
   Union[Level[Map[(* make inert *) Hold,
                    Last /@ (MapAt[Hold, #, 2]& /@
                 (* the definition of symbol *) DownValues[symbol]),
          {-1}, Heads -> True], {-2}, Heads -> True]] /.
                 f_[] :> f, Hold[_?NumberQ] | Hold[_String]],
            ((Context @@ #) =!= "System`")&]
```

```
SetAttributes[calledFunctionsLevel1, HoldAll];

(* the functions symbol depends on *)
calledFunctionsLevel1[symbol_] :=
        symbol ⟹ Select[symbolsUsed[symbol], (symbolsUsed[#] =!= {})&]

SetAttributes[dependencies, HoldAll];

(* a list of recursive dependencies *)
dependencies[symbol_] :=
Module[{dependentLevel, functionBag, i, newFunctions},
        dependentLevel[1] = {calledFunctionsLevel1[symbol]};
 (* all functions already encountered *)
 functionBag = Last /@ dependentLevel[1];
 i = 1;
 While[(* the functions of the next level *)
       dependentLevel[i + 1] = calledFunctionsLevel1 /@
                    Union[Flatten[Last /@ dependentLevel[i]]];
       (* new encountered functions *)
       newFunctions = Union[Flatten[Last /@ dependentLevel[i + 1]]];
       (* still newly encountered functions? *)
       Complement[newFunctions, functionBag] =!= {} && i < 4,
       (* the functions of the next level *)
       dependentLevel[i + 1] = Complement[dependentLevel[i + 1],
                     Flatten[Table[dependentLevel[k], {k, i}]]];
       i = i + 1;
       (* update  functionBag *)
       functionBag = Union[Flatten[{functionBag, dependentLevel[i + 1]}]]];
       (* remove empty lists *)
       DeleteCases[DeleteCases[
         Table[dependentLevel[k], {k, i + 1}],
                 (* format output *)  (_ ⟹ {}) | (a_ ⟹ {a_}),
                     Infinity], {}, Infinity] /. Hold -> HoldForm]
```

Here is a simple example of how the function `dependencies` works.

```
Clear["f*", "g*", x];
f1[x_] := f2[x] + f3[x^3 + f4[x]];
f2[x_] := g2[x] + f3[x + x^3];
f3[x_] := g2[x + f4[-x]];
f4[x_] := -Log[x] + g2[Tan[x]];
g2[x_] := x

dependencies["f1"]
```

Here is a recursive definition (to make it useful, it should be supplemented with initial conditions).

```
Clear["𝕗*"];
𝕗1[x_] := 𝕗2[x]
𝕗2[x_] := 𝕗1[x - 1]
```

In this case, `Dependencies` continues to analyze the dependencies until it finds a “closed loop”.

```
dependencies["𝕗1"]
```

Let us have a look at two examples, the commands `ContourPlot3D` from the *Mathematica* packages and the `ChapterOverview` from the package generating the chapter overviews.

```
(* turn off messages caused by usage message names *)
Off[Context::"notfound"]
Off[DownValues::"sym"]
```

```
Needs["Graphics`ContourPlot3D`"]

dependencies["ContourPlot3D"]
```

To make the last output more easily readable, we remove the long context specifications.

```
removeContexts[HoldForm[f_]] :=
Module[{fString = ToString[f], pos},
          (* rightmost position of context marker ` *)
          pos = Max[StringPosition[fString, "`"]];
      If[pos > 0, StringTake[fString, {pos + 1,
                        StringLength[fString]}], fString]]

%% /. HoldForm[f_] :> removeContexts[HoldForm[f]]
```

Here is another example, the function `PolynomialContinuedFraction` from the package `Algebra`Polyno⋱ mialContinuedFractions``. This time, we change the context to get a short output.

```
Needs["Algebra`PolynomialContinuedFractions`"]

ClearAttributes[PolynomialContinuedFraction, ReadProtected]

Begin["Algebra`PolynomialContinuedFractions`Private`"];
dependencies["PolynomialContinuedFraction"]
End[];
```

Finally, let us apply our function `dependencies` to the `ChapterOverview` from the package generating the chapter overviews.

```
Get[ToFileName[ReplacePart[
              "FileName" /. NotebookInformation[EvaluationNotebook[]],
              "ChapterOverview.m", 2]]];

dependencies["ChapterOverview"]
```

A more detailed treatment of dependencies can be found in [314✱]. We end such investigations here and invite the reader to continue in this direction. For some other similar investigations, see [195✱].

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

■ 6.4.3 More General Operations

The more general operations include matrix multiplication and the computation of inner and outer products. Although these operations belong with the mathematical operations in Subsection 6.5.1, we discuss them here because they can be used to perform more general operations on nested lists (and we will use them from time to time for programming issues; and not only in the mathematical sense). We begin with matrix multiplication.

`Dot[`$list_1$`, `$list_2$`, … , `$list_n$`]`
or
$list_1 . list_2 . \cdots . list_n$

gives the result of matrix multiplication of the lists $list_i$. For deeply nested lists, the last index of the left argument is paired with the first index of the right argument. Multiplication is carried out from the right to the left.

This definition of matrix multiplication (pairing the last index of the left list with the first index of the right list) makes it unnecessary to differentiate between row and column vectors.

Here is the result of multiplying three matrices `mata`, `matb`, and `matc` together.

```
mata = Array[a, {3, 4}]

matb = Array[b, {4, 2}]

matc = Array[c, 2]

mata.matb.matc
```

Here, we group the factors in two different ways.

```
mata.Identity[matb.matc] - Identity[mata.matb].matc // Expand
```

Here is the scalar product of two vectors.

```
{ax, ay, az}.{bx, by, bz}
```

Their length does not have to be 3, of course.

```
{ax1, ax2, ax3, ax4}.{bx1, bx2, bx3, bx4}
```

Here are three rotation matrices. `ℛx` rotates around the *x*-axis, `ℛy` rotates around the *y*-axis, and `ℛz` rotates around the *z*-axis.

```
ℛx[φ_] = {{1, 0, 0}, {0, Cos[φ], Sin[φ]}, {0, -Sin[φ], Cos[φ]}};
ℛy[φ_] = {{Cos[φ], 0, Sin[φ]}, {0, 1, 0}, {-Sin[φ], 0, Cos[φ]}};
ℛz[φ_] = {{Cos[φ], Sin[φ], 0}, {-Sin[φ], Cos[φ], 0}, {0, 0, 1}};
```

This is the result of applying three rotations to the vector `{ξ, η, ζ}`.

```
vec1 = (ℛx[φx].ℛy[φy].ℛz[φz]).{ξ, η, ζ}
```

The order of the rotation matters. We see this, for instance, by giving specialized values for the parameters involved.

```
vec1 - (ℛy[φy].ℛx[φx].ℛz[φz]).{ξ, η, ζ} /.
          {ξ -> 1, η -> 0, ζ -> 0, φx -> Pi/2, φy -> -Pi/2, φz -> Pi}
```

The norm of a vector is an invariant under rotations.

```
vec1.vec1 // Simplify
```

`Dot` represents the scalar product, and the vector product is calculated in *Mathematica* using `Cross`.

`Cross[`$list_1$`,` $list_2$`, … ,` $list_n$`]`
or
$list_1 \times list_2 \times \cdots \times list_n$
gives the vector product of the lists $list_i$. To be well-defined, the length of the lists $list_i$ must be $n+1$.

Here is the cross product between two symbolic vectors in $\mathbb{R}^3$.

```
Cross[{ax, ay, az}, {bx, by, bz}]
```

A useful application, especially for graphics, is the following representation of rotating the point *point* by an angle φ around an axis through the origin with components *dir* [253✱], [233✱].

```
rotation[point_, dir_, φ_] :=
Cos[φ] point + (1 - Cos[φ]) point.dir dir + Sin[φ] Cross[dir, point]
```

A rotation does not change distances between points. (Here we use `Simplify` with a second argument; see Chapter 1 of the Symbolics volume [303✱] for details.)

```
Module[{P1, P2, x1, y1, z1, x2, y2, z2, dx, dy, dz, φ},
(* original points *)
{P1, P2} = {{x1, y1, z1}, {x2, y2, z2}};
(* rotated points *)
P1a = rotation[P1, {dx, dy, dz}, φ];
P2a = rotation[P2, {dx, dy, dz}, φ];
(* simplified difference of distances *)
Simplify[(P1a - P2a).(P1a - P2a) - (P1 - P2).(P1 - P2),
         (* dir is a unit vector *) {dx, dy, dz}.{dx, dy, dz} == 1]]
```

The ordinary cross product in three dimensions, typically viewed as the "upper half" of an antisymmetrical tensor of rank 2, can be generalized to n dimensions [47✱], [158✱], [273✱], [125✱], [106✱], [315✱], [278✱], [244✱], and [74✱]. In $\mathbb{R}^n$, the cross product is a function of $n-1$ vectors. Here are some examples for $n = 2$ and $n = 4$.

```
Cross[{a1, a2}]
```

```
Cross[{a1, a2, a3, a4}, {b1, b2, b3, b4}, {c1, c2, c3, c4}]
```

The cross product `Cross` for the $n-1$ n-dimensional vectors $\vec{a}_1, \vec{a}_2, \ldots, \vec{a}_{n-1}$ has the following properties:

- $\vec{a}_1 \times \vec{a}_2 \times \cdots \times \vec{a}_{n-1}$ is a vector in $\mathbb{R}^n$
- $\vec{a}_1 \times \vec{a}_2 \times \cdots \times \vec{a}_{n-1}$ is orthogonal to each of the $\vec{a}_i$, $i = 1, \ldots n-1$
- $\vec{a}_1 \times \vec{a}_2 \times \cdots \times \vec{a}_{n-1}$=0 if and only if the $\vec{a}_i$, $i = 1, \ldots n-1$ are linear dependent
- $|\vec{a}_1 \times \vec{a}_2 \times \cdots \times \vec{a}_{n-1}|$ is the volume of the parallelotope formed by the $\vec{a}_i$, $i = 1, \ldots n-1$
- $\vec{a}_1 \times \vec{a}_2 \times \cdots \times \vec{a}_{n-1}$ is completely antisymmetric

For another possible generalization between the cross product of two tensors in $\mathbb{R}^n$, see [102✱].

A very useful generalization of the inner product (`Dot` or scalar product) is `Inner`.

`Inner[`*timesSynonym*`,` *list*$_1$`,` *list*$_2$`,` *plusSynonym*`]`

gives the scalar product of *list*$_1$ with *list*$_2$, but with multiplication replaced by *timesSynonym*, and addition replaced by *plusSynonym*. The head of *list*$_1$ and *list*$_2$ need not be `List`. If *list*$_1$ and *list*$_2$ contain nested expressions with the same head, the last index of *list*$_1$ is paired with the first index of *list*$_2$.

The usual scalar product is obtained as follows.

```
Inner[Times, Array[a, 6], Array[b, 6], Plus]
```

The usual matrix multiplication can also be done with `Inner`.

```
Inner[Times, {{a, b}, {c, d}}, {x, y}, Plus]
```

The second and third arguments of `Inner` do not have to have the head `List`, but they must have the same head.

```
Inner[Plus, nis[1, 2, 3, 4, 5], nis[1, 2, 3, 4, 5], soc]
```

```
Inner[Plus, nis[1, 2, 3, 4, 5], nies[1, 2, 3, 4, 5], soc]
```

A very useful generalization of the outer product that is known from matrix theory is `Outer`.

`Outer[`*timesSynonym*`,` $list_1$`,` $list_2$`, …,` $list_n$`]`

gives the outer (Kronecker) product of the lists $list_1$ with $list_2$… , but with multiplication replaced by *timesSynonym*. The head of $list_1$ and $list_2$ need not be `List`.

`Outer[`*timesSynonym*`,` $list_1$`,` $list_2$`, …,` $list_n$`,` *maxLevel*`]`

results in outer multiplication down to level *maxLevel*.

The usual outer product arises if we choose `Times` for *timesSynonym*. (It essentially amounts to replacing each element of `Array[a, 3]` by that element times a copy of `Array[b, 3]`.)

```
Outer[Times, Array[a, 3], Array[b, 3]]
```

In the following generalization, we replace `Times` by 𝕋.

```
Outer[𝕋, Array[a, 3], Array[b, 2]]
```

If higher dimensional objects are multiplied together using `Outer`, the resulting brackets may not be as expected.

```
Outer[List, Array[a, {2, 2}], Array[b, {2, 2}]]
```

With nested `Lists` as arguments in `Outer`, we often want the outer product to be calculated only on the first level. This result can be achieved by using the optional fourth argument of `Outer`.

```
Outer[CFD, {{1, 1}, {2, 2}}, {{2, 2}, {3, 3}}, 1]
```

To apply a function to elements with the same indices in several lists, we can use `Thread`.

`Thread[`*function*`[`$list_1$`,` $list_2$`, …,` $list_n$`]]`

evaluates to
`{`*function*`[`$list_1$`[[1]],` $list_2$`[[1]], …,` $list_n$`[[1]]], …,`
function`[`$list_1$`[[2]],` $list_2$`[[2]], …,` $list_n$`[[2]]], … }`

The head of $list_i$ need not be `List`, but in this case, the corresponding head should be inserted as the second argument as in `Thread[`*function*`[`$head_1$`,` $head_2$`, …,` $head_n$`],` *head*`]`.
`Thread[`*expr*`]` combines only the first level of the list *expr*.

Here is a simple example in which `Thread` causes `f` to operate on groups of arguments with the same index.

```
f[Table[a[i], {i, 4}], Table[b[i], {i, 4}], Table[c[i], {i, 4}]]
```

```
Thread[%]
```

If the argument of `Thread` is a matrix, the result `Thread[`*m*`]` is the transposed matrix.

With a head other than `List`, a second argument is needed in `Thread`.

```
Thread[f[list[a[1], a[2], a[3], a[4]],
         list[b[1], b[2], b[3], b[4]],
         list[c[1], c[2], c[3], c[4]]]]
```

```
Thread[f[list[a[1], a[2], a[3], a[4]],
         list[b[1], b[2], b[3], b[4]],
         list[c[1], c[2], c[3], c[4]]], list]
```

In the next example, `f` has only one argument, which consists of three sublists.

```
Thread[f[{Table[a[i], {i, 4}], Table[b[i], {i, 4}], Table[c[i], {i, 4}]}]]
```

The expression remains unchanged. If the function *function* has the attribute `Listable`, the operation carried out by `Thread` automatically takes place for arguments with depth 1.

In the following example, we attempt to provide the function `triangle` with three vertices taken from three lists containing the coordinates of the first, second, and third vertices of five triangles.

```
triangle[Table[vertex1[i], {i, 5}],
         Table[vertex2[i], {i, 5}],
         Table[vertex3[i], {i, 5}]]
```

Now, we create five triangles, each with three vertices.

```
Thread[%]
```

If the lists have more depth, a difference between `Thread` and using the attribute `Listable` exists.

```
Remove[fg];
SetAttributes[fg, Listable];
fg[{{1, 11}, {2, 22}}, {{a1, a2}, {b1, b2}}]
```

The attribute `Listable` works for arbitrary depth, whereas `Thread` works only at level 1.

```
Remove[fg];
Thread[fg[{{1, 11}, {2, 22}}, {{a1, a2}, {b1, b2}}]]
```

The following command is closely related to the command `Thread`.

`MapThread[`*function*`, {`$list_1$`, `$list_2$`, …, `$list_n$`}, `*levelSpecification*`]`
applies *function* to corresponding elements in the lists $list_i$ at level *levelSpecification*. The head of $list_i$ need not be `List`.

In the next two examples, `MapThread` gives the same results as `Thread`.

```
MapThread[f, {Array[a, {4}], Array[b, {4}], Array[c, {4}]}]
```

```
Clear[f, a, b, c];
MapThread[f, {{Array[a, {4}], Array[b, {4}], Array[c, {4}]}}]
```

But the following example would not be possible using `Thread`.

```
MapThread[ϒ, {{{1, 11}, {2, 22}}, {{a1, a2}, {b1, b2}}}, 2]
```

We could have gotten the same result in the last example by assigning the attribute `Listable` to `f`; however, not all of the following examples could be done this way because we could not control the level specification in this case. We use all four sensible level specifications.

```
MapThread[ϒ, {Array[a, {3, 3, 3}]}, 0]
```

```
MapThread[ϒ, {Array[a, {3, 3, 3}]}, 1]
```

```
MapThread[ϒ, {Array[a, {3, 3, 3}]}, 2]
```

```
MapThread[ϒ, {Array[a, {3, 3, 3}]}, 3]
```

Here is a somewhat different example of the application of `MapThread`. Suppose we are given two matrices: a matrix whose elements are operators and a matrix whose elements are the associated arguments. This is a matrix of functions.

```
operatorMatrix = Table[Evaluate[i + j + #]&, {i, 3}, {j, 2}]
```

And this is a matrix of arguments.

```
argumentMatrix = Table[i + j, {i, 3}, {j, 2}]
```

Now, each operator is to be applied to "its" argument. The following input does not work, of course.

```
operatorMatrix[argumentMatrix]
```

But this input does.

```
MapThread[#1[#2]&, {operatorMatrix, argumentMatrix}, 2]
```

`Distribute` is one more important command that belongs in this section.

`Distribute[`*expression*`,` *whatOver*`,` *what*`,` *whatOverNew*`,` *whatNew*`]`

applies the distributive law for *whatOver* to *what* in *expression*, and then replaces the head *what* by the head *whatNew* and the head *whatOver* by the head *whatOverNew*. The third, fourth, and fifth arguments need not appear; in this case, the heads are not changed.

Here are two examples of this relatively abstract command.

```
Distribute[wo[wa[a1, a2, a3, a4], wa[b1, b2, b3, b4]], wa]
```

```
Distribute[wo[wa[a1, a2, a3, a4], wa[b1, b2, b3, b4]],
           wa, wo, WA, WO]
```

Here are the fourth and fifth arguments in `Distribute` symbols.

```
Distribute[wo[wa[a1, a2, a3, a4], wa[b1, b2, b3, b4]],
           wa, wo, WA[1], WO]
```

In the next input, the fourth and fifth arguments in `Distribute` are pure functions.

```
Distribute[wo[wa[a1, a2, a3, a4], wa[b1, b2, b3, b4]],
           wa, wo, WA[##]&, WO[#]&]
```

We will make considerable use of the commands `Thread`, `Apply`, `Map`, `Inner`, and `Distribute` later in dealing with graphics (we have already used `Distribute` in the beginning graphic in the first chapter).

Let us finish this subsection by reiterating the importance of the list-manipulating functions discussed so far in this chapter.

Whenever possible, manipulations on lists should always be done on the entire list(s) (i.e., using the commands `Map`, `MapThread`, `Thread`, `Apply`, `Inner`, `Outer`, `Distribute`, etc.), rather than on one element at a time, which leads to a great savings in computational time.

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

■ 6.4.4 Constructing a Crossword Puzzle

In this subsection, we examine a problem that extensively involves lists. Suppose we are given a list of words (which, without loss of generality, we can assume are in the form of lists of their letters, i.e., strings). The aim is to insert them in a rectangular grid in such a way that each word is either horizontal (reading from left to right) or vertical (reading from top to bottom), and such that words are connected at subrectangles containing a common letter. (A subrectangle is the space occupied by one letter.) Here is an example.

```
              D
              u
 H        Salzmann               I
 ö           p       G          n  B
 r     F      l   Eisbein      Eisenach
 s     r   Schiller   r   F A W  e  c
 e     i      n   f   m   o u a  l  h
 l    Rennsteig   u Bratwurst r  s
 b     d      s   r   n   e o t  b
Tenneberg         t   y   s b b Weimar
 r     i                  t a u  r
 g     c                   Thuringia
       h                    n g
       r
      Gotha
       d
      Walterhausen
```

For better readability, we assume that any two horizontal words and any two vertical words are separated by at least one blank space. In addition, no horizontal word should begin or end in a subrectangle, which is next to one occupied by a letter in a vertical word, and no vertical word should begin or end in a subrectangle, which is next to one occupied by a letter in a horizontal word. However, we do allow a word to begin or end in a subrectangle, which is occupied by another letter.

In the following code, we do not protect all variables which arise, but instead use the following variables globally throughout this section: (`v` always indicates vertical and `h` always indicates horizontal.)

■ `placed`: This is a list of the words that have already been placed in the puzzle in the form {*coordinates*`,` *stringOfLetters*}.

■ ω: ω`[`i`]` contains the positions of those letters of the ith placed word where another word can be joined.

■ `closed`: `closed["h"]` and `closed["v"]` contain the coordinates of those cells that cannot be occupied by letters of words to be placed horizontally or vertically, respectively.

■ `startClosed` and `endClosed`: The lists `startClosed["h"]`, `startClosed["v"]`, `endClosed["h"]`, and `endClosed["v"]` contain the coordinates of those cells that cannot be used for the starting or ending letters of words to be placed horizontally or vertically, respectively.

■ `words`: This list contains the words that have not yet been used.

The idea of the implementation is as follows. We choose an initial word and an initial direction. Then, we take the next word and check to see if it can be attached to any of the previously placed words. If not, we check the next word, and so on, at each step making sure that none of its letters fall in a closed cell, and that it also fits with the other words.

We begin by initializing the lists `placed`, `closed["h"]`, `closed["v"]`, `startClosed["h"]`, `startClosed["v"]`, `endClosed["h"]`, and `endClosed["h"]`, as well as ω`[1]`. This initialization is done by the function `initialization`; its argument is a list of the letters of the first word and its orientation (`"h"` for horizontal or `"v"` for vertical).

For the sake of efficiency, we do not bother to carry out tests on the arguments for correctness in the following auxiliary routines, although we do include tests in the final function `crossWordConstruction`.

```
initialization[start_] :=
Module[{data, wt},
(* is the first word aligned horizontally? *)
wt = (start[[2]] === "h");
(* set the letters of the first words *)
placed = MapThread[List, {data =
If[wt, Table[{i, 1}, {i, Length[start[[1]]]}],
       Table[{1, i}, {i, 1, -Length[start[[1]]] + 2, -1}]], start[[1]]}];
(* vertical and horizontal letters to be set *)
Clear[ω];
ω[1] = {placed, start[[2]]};
closed["h"] =
 If[wt, Union[{First[#] + {-1, 0}}, {Last[#] + {+1, 0}},
             # + {0, 1}& /@ #, # + {0, -1}& /@ #],
(* ω[1][[2]] === "v" *)
{First[#] + {0, 1}, Last[#] + {0, -1}}]&[data];
closed["v"] =
 If[wt, {First[#] + {-1, 0}, Last[#] + {+1, 0}},
(* ω[1][[2]] === "v" *)
     Union[{First[#] + {0, 1}}, {Last[#] + {0, -1}},
            # + {1, 0}& /@ #, # + {-1, 0}& /@ #]]&[data];
(* the spaces which cannot be occupied any more *)
startClosed["v"] = If[ wt, # + { 0, -1}& /@ data, {}];
endClosed[  "v"] = If[ wt, # + { 0,  1}& /@ data, {}];
startClosed["h"] = If[!wt, # + { 1,  0}& /@ data, {}];
endClosed[  "h"] = If[!wt, # + {-1,  0}& /@ data, {}]; ]
```

We now look at an example.

```
initialization[{{"A", "b", "o", "r", "t"}, "h"}]
```

The various lists have the following values. The list `placed` contains the letters of the first word, which is horizontal and starts at {0, 0}.

```
placed
```

The letters of other horizontal words cannot be placed in the cells immediately over, under, and next to the letters of this first word.

```
closed["h"]
```

No word written vertically can pass through the cells directly to the left of `A` and directly to the right of `t`.

```
closed["v"]
```

A word written horizontally has no influence (except via `closed["h"]`) on the positions of the first and last letters of other words written horizontally.

```
startClosed["h"]; endClosed["h"];
```

Vertical words may not begin directly below letters of horizontal words.

```
startClosed["v"]
```

They may not end directly above them.

```
endClosed["v"]
```

The next word can be attached to the first word at any matching letter of ω.

```
??ω
```

Next, we discuss “attaching” a word. The function `attach` takes two arguments and generates a list of lists of the form {*i*, *j*}. Each such element indicates that the *i*th letter of the word *old* matches the *j*th letter of the word *new* (where we distinguish between lowercase and uppercase letters). This list is calculated by looking at the letters that are contained in *old* as well as in *new*.

```
attach[old_, new_] :=
Sort[Flatten[Flatten[Outer[List, Sequence @@ #], 1]& /@
    (({Flatten[Position[old, #]],
       Flatten[Position[new, #]]}& /@ #)&[(* the common letters *)
                     Intersection[old, new]]), 1]]
(*  to get different word orders, we could
   randomly permute this list, e.g. with
   //  Function[y, Nest[Function[x, Function[b,
       {DeleteCases[x[[1]], Evaluate[b]],
       Flatten[{x[[2]], {b}}, 1]}][
         x[[1, Random[Integer, {1, Length[x[[1]]]}]]]]],
                    {y, {}}, Length[y]][[2]]]
*)
```

We again look at an example of joined items (from http://specialfunctions.com).

```
attach[Characters["Mathematica"], Characters["SpecialFunctions"]]
```

When the two words do not have a common letter, `attach` returns an empty list.

```
attach[Characters["Abort"], Characters["Sech"]]

attach[{}, Characters["Sech"]]
```

Suppose that `attach` has found a cell in *old* to which the word *new* may be attached. Now, we have to check whether all of the letters contained in the new word fit in the allowed space available (i.e., that none of them would fall in a cell in the closed lists, or would intersect some other word without matching letters). We accomplish this result with the function `fits¿`. Its first argument is ω[*i*] (i.e., a list of those letters of a word that have already been placed where the new word can be attached, along with its orientation `"h"` or `"v"`). Its second argument is the new word *new*, and its third argument *combi* is one of the possibilities in the output of `attach` that lists the ways of attaching the new word to the old (i.e., which letter of the first word can be attached to which letter of the old one). The cells needed to place the word are contained in `cellsNeeded`. Depending on the orientation of the old word (horizontal or vertical), the new word is attached correspondingly (vertical or horizontal).

```
fits¿[oldω_, new_, combi_] :=
Module[{wt, cellsNeeded, lettersNeeded, occupiedCells, h1, wt1},
(* was old word aligned horizontally or vertically? *)
wt = oldω[[2]] === "h"; wt1 = If[wt, "v", "h"];
(* which cells with which letters are needed *)
 If[wt, h1 = oldω[[1, combi[[1]], 1]] (* Starting point *);
   cellsNeeded = Table[h1 + {0, j},
         {j, combi[[2]] - 1, -(Length[new] - combi[[2]]), -1}],
     (* oldω[[2]] === "v" *)
   h1 = oldω[[1, combi[[1]], 1]];
   cellsNeeded = Table[{j, 0} + h1,
         {j, -combi[[2]] + 1, Length[new] - combi[[2]]}]; ];
  (* test if the new word would have any letters in cells
  that are already occupied or are closed *)
   yesNo =
   And[Intersection[{First[cellsNeeded]}, startClosed[wt1]] === {},
       Intersection[{Last[cellsNeeded]}, endClosed[wt1]] === {},
       Intersection[cellsNeeded, closed[wt1]] === {},
       (* could use hashing instead of list operations *)
       occupiedCells = Cases[placed, Evaluate[Alternatives @@ ({#, _}& /@
                                                              cellsNeeded)]];
     (* check if the letters fits also into other already
    placed words if they overlap *)
       lettersNeeded = MapThread[List, {cellsNeeded, new}];
       Complement[occupiedCells, lettersNeeded] === {}
       (* Here, additional conditions on the directions
    of growth and restrictions on the domain
    could be implemented. *)];

{yesNo, If[yesNo, {lettersNeeded, occupiedCells}, Null]}]
```

If it is possible to attach *new* to *oldω*, `fits¿` produces a list of the form `{True, {`*lettersNeeded*`,`*occupiedCells*`}}`, and if it is not possible, it gives `{False, Null}`. (Because `fits¿` does not only give `True` or `False` as a result, we do not let it end with `Q`, but rather with `¿`.) Here, two places exist where the new word fits the current `ω[1]`.

```
ω[1]

fits¿[ω[1], {"$", "A", "b", "o", "r", "t", "e", "d"}, {1, 2}]

fits¿[ω[1], {"$", "A", "b", "o", "r", "t", "e", "d"}, {2, 3}]
```

Next, we repeatedly apply `fits¿` to the results of `attach` until some suitable fit is found (if any exists). This process is accomplished with `combiSearch`. The arguments for `combiSearch` (except for the third one, which is not needed here) are the same as those for `fits¿`.

```
combiSearch[oldω_, new_] :=
Module[{combinations, tempData, i},
combinations = attach[Last /@ oldω[[1]], new];
(* try other combination to fit new *)
If[combinations =!= {},
For[i = 1, (* until it fits *)
     ((i <= Length[combinations]) &&
     !(tempData = fits¿[oldω, new, combinations[[i]]])[[1]]),
     i = i + 1,
     Null];
If[tempData[[1]] === True,
    (* the new cells to be occupied and their content *)
   {tempData[[2, 1]], tempData[[2, 2]], oldω[[2]]}, $Failed],
    (* impossible to continue *) $Failed]]
```

We again look at the example above. The first fit is returned in the form {*lettersNeeded*, *occupiedCells*}.

```
combiSearch[ω[1], {"$", "A", "b", "o", "r", "t", "e", "d"}]
```

If no fit can be found, for example, if `oldω` and `new` do not have any common letters (or if all cells are already occupied), `combiSearch` returns `$Failed`.

```
combiSearch[ω[1], {"T", "Z", "u", "i"}]
```

Once we have found a possible configuration for attaching a word, we now have to carry out the attachment; that is, the letters of the new word have to be put into the list `placed`, and the lists `closed["h"]`, `closed["v"]`, `startClosed["h"]`, `startClosed["v"]`, `endClosed["h"]`, and `endClosed["h"]` have to be updated. In addition, a new definition is needed for v, and the attachment cell along with its immediate neighbors (important for efficiency) have to be removed from the list of allowable attachment points for previously placed words. To avoid the tiresome process of looking through all relevant words, this is done by `attachAndUpdate` by immediately changing the definition of all ωs via `DownValues[ω] = DeleteCases[DownValues[ω], Evaluate[remove], {4}]`.
The arguments of `attachAndUpdate` are simply the values produced by `combiSearch`.

```
attachAndUpdate[{newLetters_, common_, oldHOrV_}] :=
Module[{remove, letterCells},
 (* all placed letters *)
 placed = Join[placed, Complement[newLetters, common]];
 letterCells = First /@ newLetters;
(* keep all global lists updated *)
 If[oldHOrV === "h",
    startClosed["h"] =
     Union[startClosed["h"], # + {1, 0}& /@ letterCells];
   endClosed["h"] =
    Union[endClosed["h"], # + {-1, 0}& /@ letterCells];
   closed["h"] = Union[closed["h"],
            Union[{First[letterCells] + {0, 1}},
                  {Last[letterCells] + {0, -1}}]];
   closed["v"] = Union[closed["v"],
    Union[{First[letterCells] + {0, 1}}, {Last[letterCells] + {0, -1}},
          {-1, 0} + #& /@ letterCells, {+1, 0} + #& /@ letterCells]],
   (* oldHOrV === "v" *)
   startClosed["v"] = Union[startClosed["v"], # + {0, -1}& /@ letterCells];
   endClosed["v"] = Union[endClosed["v"], # + {0, 1}& /@ letterCells];
   closed["v"] = Union[closed["v"],
        Union[{First[letterCells] + {-1, 0}}, {Last[letterCells] + {1, 0}}]]
   closed["h"] = Union[closed["h"],
        Union[{First[letterCells] + {-1, 0}}, {Last[letterCells] + {1, 0}},
              {0, 1} + #& /@ letterCells, {0, -1} + #& /@ letterCells]]];
(* look at DownValues of v and manipulate them directly *)
ω[Length[DownValues[ω]] + 1] = {newLetters, If[oldHOrV === "h", "v", "h"]};
(* no longer possible positions to join a word *)
remove = Alternatives @@ ({#, _}& /@
      Join[#, {0, 1} + #& /@ #, {0, -1} + #& /@ #,
              {1, 0} + #& /@ #, {-1, 0} + #& /@ #]&[First /@ common]);
(* the new DownValues for v *)
DownValues[ω] = DeleteCases[DownValues[ω],
                     Evaluate[remove], {4}]; ]
```

We now look at how the values of the global quantities are modified. We start the process anew.

```
initialization[{{"A", "b", "o", "r", "t"}, "h"}];
```

Here, the new word `$Aborted` is attached.

```
combiSearch[ω[1], {"$", "A", "b", "o", "r", "t", "e", "d"}]

attachAndUpdate[%]
```

Now, we use `placed`.

```
placed
```

These are the positions where further words can be attached.

```
??ω
```

In the following cells, no letter can ever be placed.

```
??closed

??startClosed

??endClosed
```

When `combiSearch` cannot find a fit for a given first argument, the first argument has to be changed and the search repeated. This process is done by the function `next`. Its argument is just the word *new*, which is to be attached. If a fit is found, it returns the result of the associated `combiSearch` call, and if no fit can be found, it returns `$Failed`.

```
next[newOne_] :=
Module[{maxi, res, j},
 (* how long to try *) maxi = Length[DownValues[ω]];
 For[j = 1, (* try until it fits *)
     j <= maxi && ((res = combiSearch[ω[j], newOne]) === $Failed),
     j = j + 1,
     Null]; (* or give up if it is impossible *)
           If[res =!= $Failed, res, $Failed]]
```

We demonstrate how it works.

```
initialization[{{"A", "b", "o", "r", "t"}, "h"}];

next[{"$", "A", "b", "o", "r", "t", "e", "d"}]
```

We are almost finished with our implementation of the crossword puzzle. The following function `autoSearch` goes through a given collection of words *words* (in the form of a list of their letters) until it finds one that can be attached to the previously placed words. If none is found, it gives `$Failed`.

```
autoSearch :=
Module[{counter = 0, res},
For[(* what to fit *)
     newOne = words[[1]],
     If[counter > Length[words] - 1, False,
        (res = next[newOne]) === $Failed],
     (* shift to get new constellation *)
     words = RotateLeft[words];
     newOne = words[[1]];
     counter = counter + 1,
     Null]; res]
```

We are finally ready to define `crossWordConstruction`. This function has three arguments: the initial word *startString* and its orientation, the list *workStrings* of the words to be used, and the number `num` of words to be attached from *workStrings*. The message `crossWordConstruction::cpafw` appears when it is no longer possible to attach words. In `crossWordConstruction`, we test the arguments for correctness, and keep the list `words` updated.

```
crossWordConstruction::cpafw =
"Cannot place any further words.";

CrossWordConstruction[startString:{_String, ("h" | "v")},
                      workStrings_?(VectorQ[#, (Head[#] === String)&]&),
                      num_Integer?(# > 1&)] :=
Module[{res},
  (* prepare workstring as single characters *)
  words = Characters /@ workStrings;
 (* start - initialization of all variables *)
  initialization[{Characters[startString[[1]]], startString[[2]]}];
 (* num times attach new word *)
  Do[res = autoSearch;
     If[res === $Failed,
           Message[crossWordConstruction::cpafw];
           (* emergency exit -- could be refined *)  Abort[]; Null,
           attachAndUpdate[res];
           words = Drop[words, 1]], {num}]] /; Length[workStrings] >= num
```

We now try out this code using the *Mathematica* built-in commands as our supply of words.

```
reservoir = Names["System`*"];
```

Here, we connect the first six built-in *Mathematica* commands.

```
CrossWordConstruction[{reservoir[[1]], "h"}, Take[reservoir, {2, 100}], 5];
```

Now, here are the contents of `placed`.

```
placed
```

Currently, exactly six values for ω exist.

```
??ω

Clear[ω]
```

We do not look at the closed lists because of their sizes.

```
Length /@ {closed["h"], closed["v"],
           startClosed["h"], startClosed["v"],
           endClosed["h"], endClosed["v"]}
```

In the following case, it is not possible to attach the third element of the second argument to already-connected letter chains. (We could implement a more graceful ending, but because we are mainly interested in the case of possible continuation, the `Abort[]` will do the job.)

```
CrossWordConstruction[{"Aaab", "h"}, {"Bbbc", "Cccc", "Dddd"}, 3]
```

But as many attachments as possible were made.

```
placed
```

Because `placed` is a list of coordinates of letters, it is not particularly convenient to read. Thus, we print the associated words as actual words written horizontally or vertically. The command `TableForm` is well suited for this formatting. It saves space and is much faster and more editable than a corresponding graphics approach like this one.

```
Function[placed, Show[Graphics[{
Rectangle[# - {0.46, 0.46}, # + {0.46, 0.46}]& /@
 Complement[Flatten[Table[{i, j}, Evaluate[
   Sequence @@ MapThread[Flatten[List[##]]&, {{i, j},
      {-1, 1} + #& /@ ({Min[#], Max[#]}& /@ Transpose[First /@
     placed])}]]], 1], First /@ placed], Text[StyleForm[
       #1, FontFamily ->"Courier", FontSize -> 12], #2]& @@ #& /@
        Reverse /@ placed}], AspectRatio -> Automatic, Axes -> False.
                             PlotRange -> All]][placed]
```

The following function `display` accomplishes what we want. The auxiliary function `iter` finds the region (including its boundary) containing all of the cells used. If a cell is occupied, the function `M` returns the letter in it; otherwise, `M` returns `"  "`. After building a table of letters or `"  "`, we use `TableForm` (with the option setting `TableSpacing -> {0, 0}`) to display the crossed words.

```
display :=
Module[{iter, M, i, j},
(* the iterator for the dimensions *)
iter = Reverse[MapAt[Append[#, -1]&,
  MapThread[Flatten[List[##]]&, {{j, i},
    MapAt[Reverse, {-1, 1} + #& /@ ({Min[#], Max[#]}& /@
      Transpose[First /@ placed]), {2}]}], {2}]];
(* make definitions for M *)
Apply[Set[M[#1], #2]&, placed, {1}];
(* the non letter cells *)
M[x_] = " ";
TableForm[(* it is just a Table in TableForm *)
  Table[M[{j, i}], Evaluate[Sequence @@ iter]],
        TableSpacing -> {0, 0}]]
```

We can finally look at `placed` graphically.

```
display
```

To conclude this subsection, we give a somewhat larger example where the first 50 built-in names are to be connected.

```
CrossWordConstruction[{reservoir[[1]], "h"},
                      Take[reservoir, {2, 200}], 49]; // Timing
```

Here, the current arrangement of letters in `placed` is shown.

```
display
```

Here is one more example using the *Mathematica* commands at the end of the alphabet.

```
CrossWordConstruction[{reservoir[[-1]], "v"},
      Reverse[Take[reservoir, {-200, -2}]], 49]; // Timing

display
```

The next example uses the words from the beginning of this Subsection.

```
tWords =
{"Salzmann", "Tenneberg", "Rennsteig", "Gotha", "Walterhausen",
 "Eisbein", "Bratwurst", "Thuringia", "Weimar", "Eisenach",
 "Hörselberg", "Friedrichroda", "Dumplings", "Erfurt", "Bach",
 "Germany", "Forest", "Autobahn", "Wartburg", "Inselsberg",
 "Schiller"};

CrossWordConstruction[{tWords[[-1]], "v"}, Rest[tWords], 20];
display
```

By using the list of words `tWords` in different orders, we can get many different word arrangements.

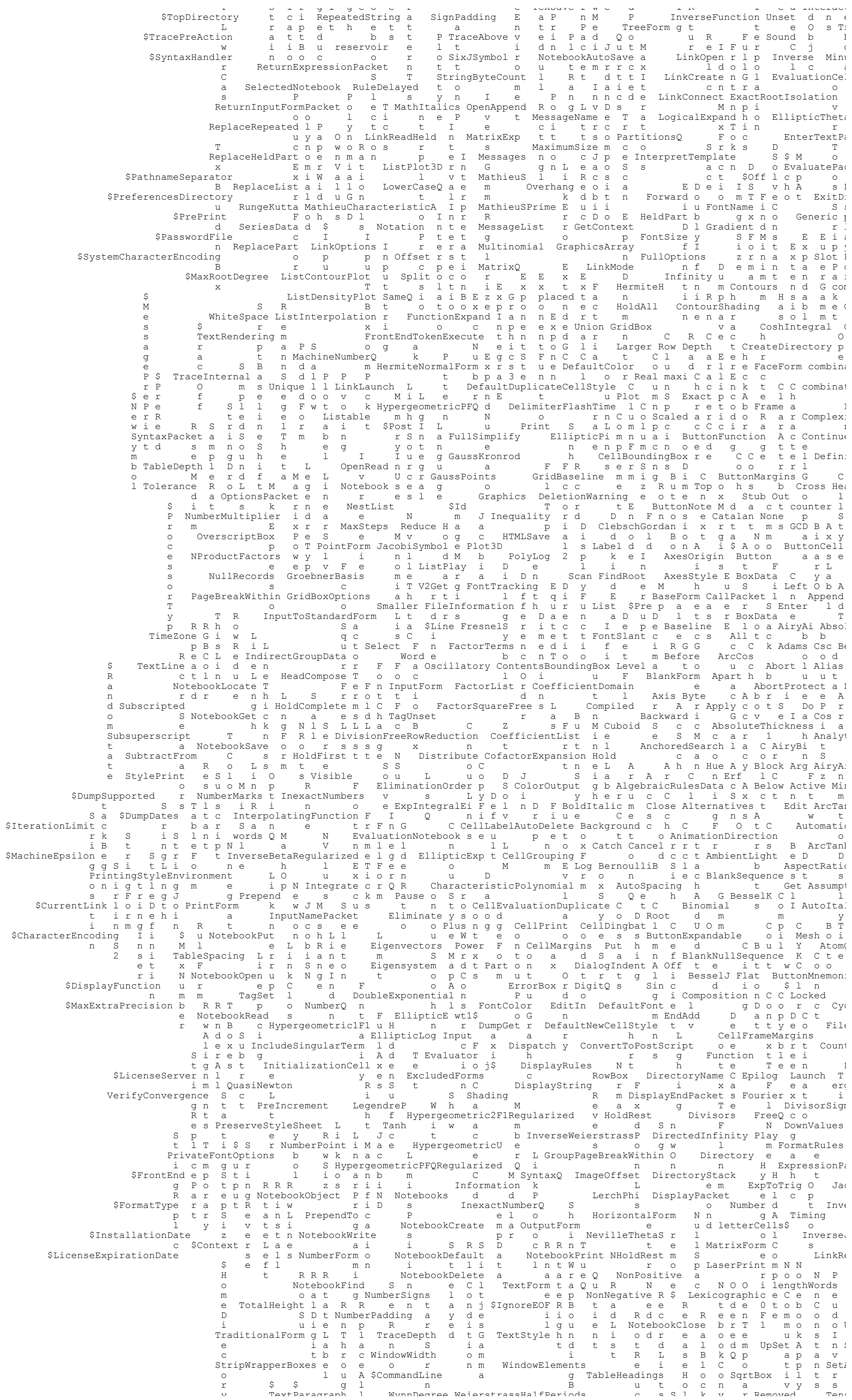
```
(* generate a random permutation of a list ℓ *)
randomPermutation[ℓ_List] :=
Module[{n = Length[ℓ], l = Range[Length[ℓ]], tmp1, tmp2, j},
    Do[(* randomly swap elements *)
       tmp1 = l[[i]]; j = Random[Integer, {i, n}]; tmp2 = l[[j]];
       l[[i]] = tmp2; l[[j]] = tmp1, {i, Length[l]}]; ℓ[[l]]]

Do[(* a random permutation of the list tWords *)
   reorderedtWords = randomPermutation[tWords];
   (* try making a crossword construction;  continue if it does not finish *)
   CheckAbort[CrossWordConstruction[{reorderedtWords[[-1]], "v"},
                         Rest[reorderedtWords], 20];
              Print @ display, Null],
   {100}]
```

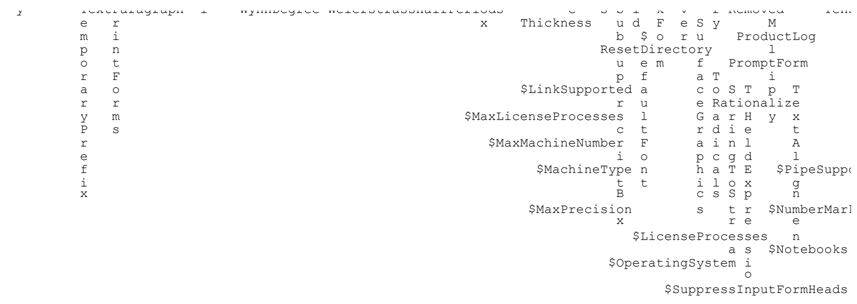
It is also possible to use all built-in *Mathematica* commands via

```
CrossWordConstruction[{reservoir[[-1]], "v"}, Rest[reservoir],
                      Length[Rest[reservoir]]];
display
```

but the computation takes longer, and the result is too large to reproduce here.

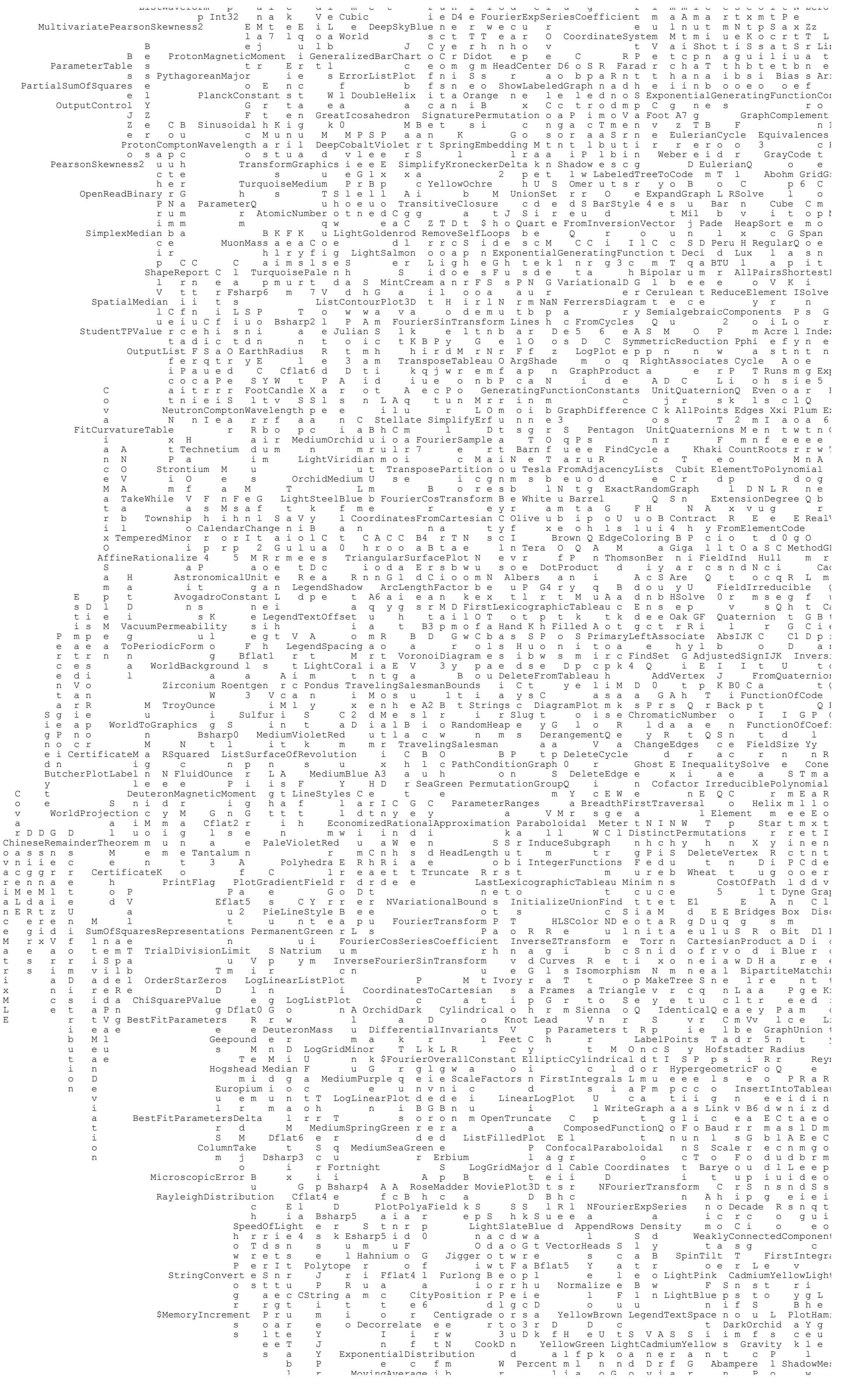
```
                                                                                               $
                                                                                               M
                                                                                               i
                                                                                               n
                                                                                               M     $    $
                                                                                               a     M    N
                                                                                               c     i    e
                                                                                               h     n $  t
                                                                                               i     P N  w
                                                                                               n     r u  o
                                                                                               e     e m  r
                                                                                               N     c b  k
                                                                                               u     i e  L
                                                                                               m     s r  i
                                                                                               b     i B  c
                                                                                               e S   o i  e
                                                                                             TargetFunctions
                                                                                                 l     s  s
                                                                                   TraditionalOrder       e
                                                                                                 c
                                                                                                 t
                                                                                             S TrigExpand
                                                                                  $ContextPath   o
                                                                                             o S n
                                                                                             w i E
                                                                                             S n v
                                                                                             p g a S
                                                                             WeierstrassPPrime l l i
                                                                                             c e u n
                                                                                             i L a g
                                                                                           S a e t u
                                                                                         S e l t e l
                                                                            workStrings  t t C t C a
                                                                                h        r F h e r r
                                                                           ViewPoint     i i a r e i
                                                                                w   S    n l r I a t
                                                                     WeierstrassP   l    g e a t t y
                                                                                a S o    B D c a e D
                                                                   WindowFloating i t    r a t l C e        Tra
                                                                                e n S  M e t e i e p
                                                               UpSetDelayed     B g e ParametricPlot    S
                                                                 e              r u q  t k   s s l h    h  Trac
                                                     TraceOriginal   S  S       e l u  h                o
                                                                 e   t  e      ParametricPlot3D      ViewCente
                                                        ViewVertical a  q       k r n  e       W S S    S
                                                                 t   c  u  S    s V c  u       h t e    t
                                                 TraditionalNotation k  e  t     ParentConnect i a t    r
                                                                 o   C  n  y      l    h       l r P  $Display
                                                              S  n   o  c  l      u   ParentDirectory   n
                                                              c  E   m  e  e      e    r         P e    g  S
                                                    TraceForward v   p  F  D      s L ParentForm r c S  C  p
                                                              i  a PolynomialLCM    U  c         o i p  h  a
                                                      TrigToExp  l   e  r  t   a    B  t      T  c s a  a  n
                                                              t OutputNamePacket   Raster  L  o  e i n  r  M
                                                            S S  a   e         h    c  r   U  R  s o A  a  a
                                                            c i  t       Orderless  k  i   D  u  s n d  c  x
                                                          S i z  e             m    S  s Overflow    j  t  S U
                                                      TraceLevel     OutputStream   u  t   c  e      u  e  i n
                                                          t n M   S            t    b  i M o  s  M   s  r  z i
                                                          OutputMathEditExpression  s  c o m M  Parenthesize n
                                                          p i l   r            c    t  E n p o   t   m       s
                                                Underscript f t   i R     Overlaps  i  x t o e SchurDecomposit
                                                          i i i WindowSize     N    t  p e s b   i   n       a
                                                          o c p   g w       S  o M  u  o C i i  Permutations l
                             T    T                       n F l   Q s PageHeight a  t  n a t u   u   s       l
                 T           a  $MachineName           S  s o i     E       n  a p  i  e r i s   C
            $ProcessID     S g    b             Unprotect P r e     q S S   I  t I  o  n l o M   P         L
                 x      S  y S    l                    r  a m r    NumericFunction InputToInputForm   NumericQ
              T  t S    u  n e  $Messages    ToUpperCase  c   s     a l g   t  o d             u i M       n
              i  J u    p  t t    A                    e  k         l e n   e InterpolatingPolynomial   F  k La
    $OutputForms u p    e  a D    l       TableDirections e           c a   g    x             c e k G  r  P
              e  s e    r  x e  $MinNumber             S  t       R   t t   r InterpolationPoints  e e  o  a
              C  t r  T s  L l    g            TotalWidth  R $Failed  a u R a    d             i   B n InputSt
$TracePostAction i s  r c  e a    n                    y   e      t R b r e l     U IntervalUnion  o e  t  t  e
              n  f c  i r  n y N  m         ThreeJSymbol   p P    u a l e a       s            n   x r  E  e  a
      T       s  i r  g i  g e o  e  P                 e  TeXSave r w e   d       i R          I   e a Interact
```

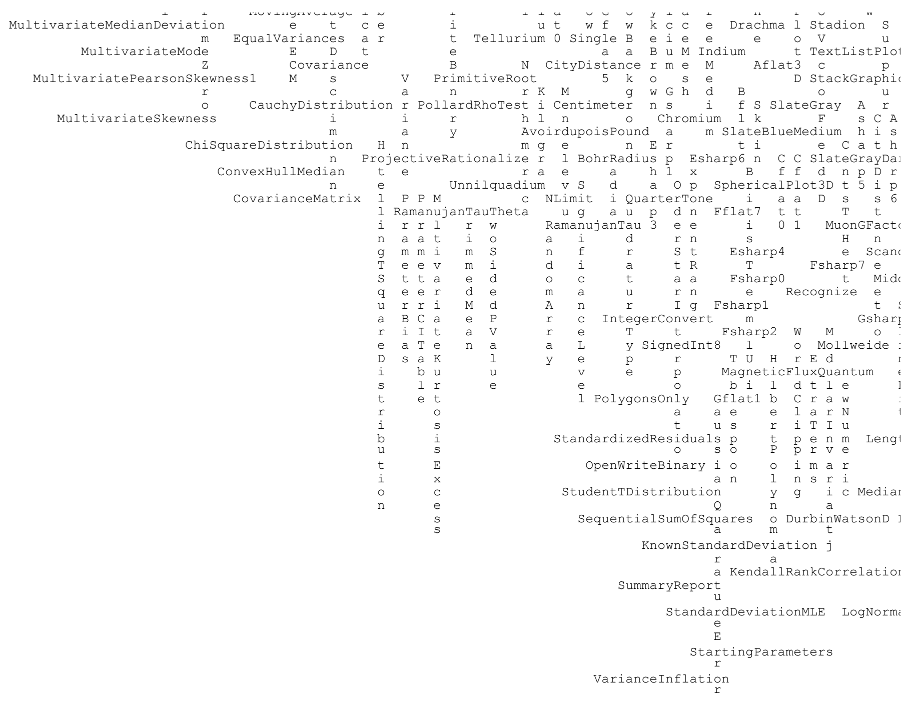




By using the command names from the packages instead of the built-in names, we could make much bigger crossword puzzles. First, many more names are available, and second, they are, on average, longer than the built-in names.

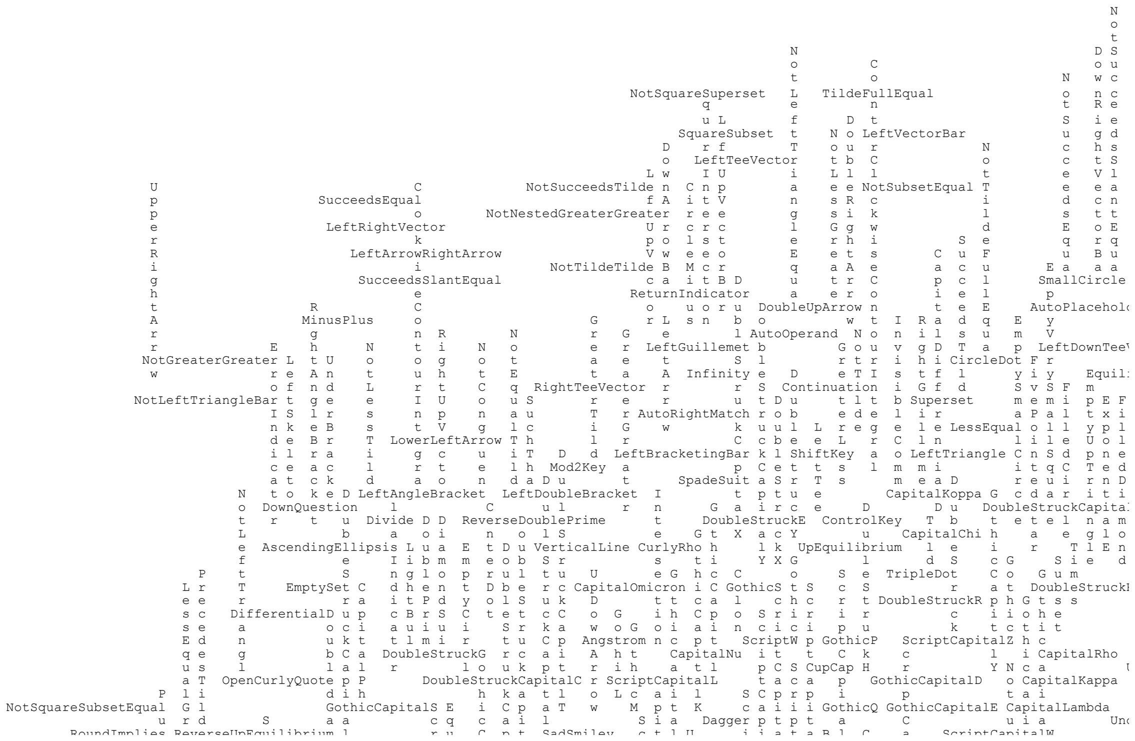
```
reservoir = Select[Complement[#2, #1#], StringLength[#] > 1&]&[
 Names["*"], Needs /@   (* all function names after package loading *)
   {"Algebra`Master`", "Calculus`Master`", "DiscreteMath`Master`",
    "Geometry`Master`", "Graphics`Master`", "LinearAlgebra`Master`",
    "Miscellaneous`Master`", "NumberTheory`Master`", "NumericalMath`Master`",
    "Statistics`Master`", "Utilities`Master`", "Utilities`Master`"};
  (* all function names before package loading *)
  Names["*"]];

CrossWordConstruction[{reservoir[[-1]], "v"}, Rest[reservoir],
                      Length[Rest[reservoir]]];
display
```

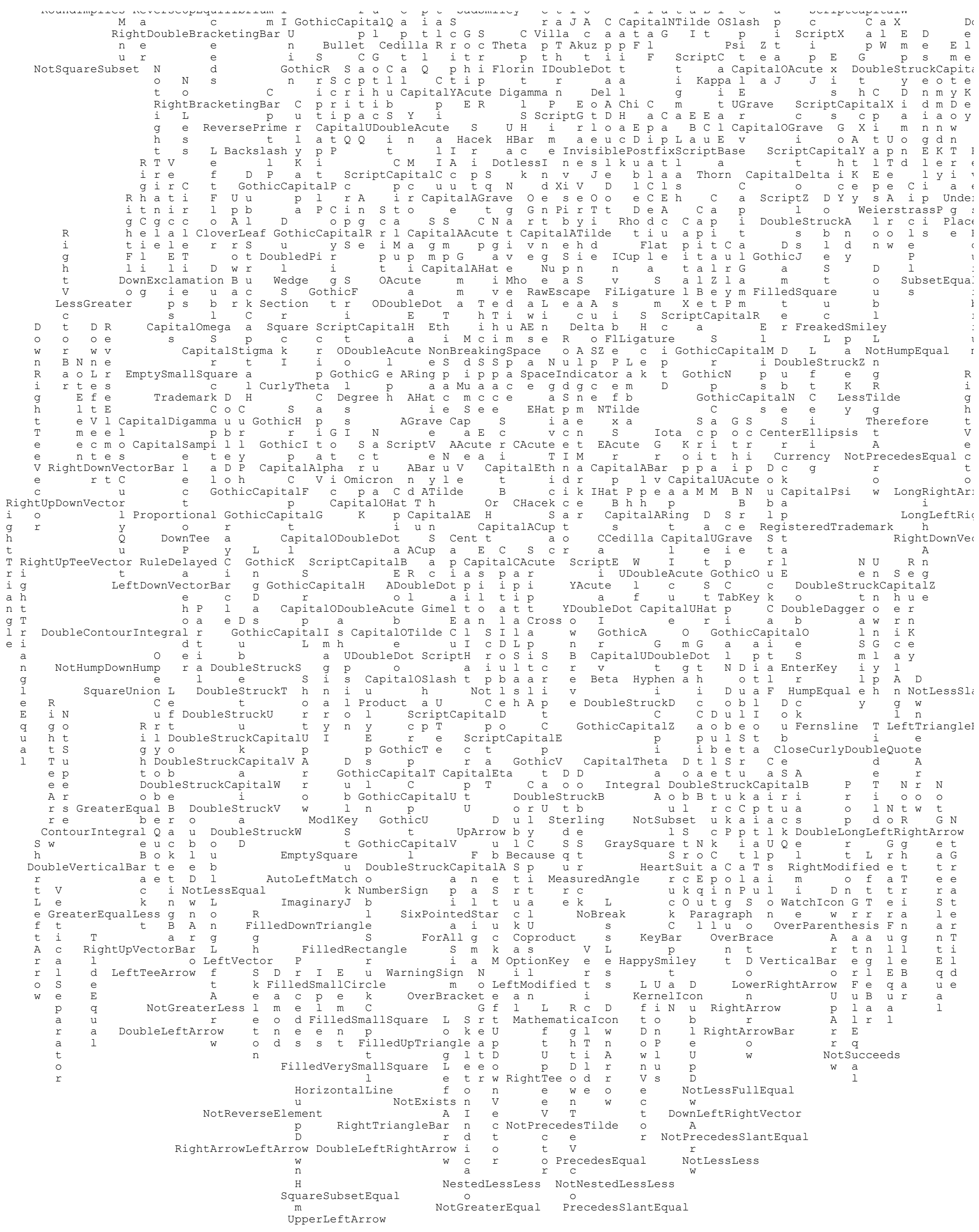




Here a different one is shown—all names of named characters in *Mathematica.*

```
reservoir = StringDrop[StringDrop[
 ToString[FullForm[#]], -2], 3]& /@
  DeleteCases[Select[FromCharacterCode /@ Range[10^5],
   Characters[ToString[FullForm[#]]][[-2]] === "]"&],"]"];

CrossWordConstruction[{reservoir[[-1]], "v"}, Rest[reservoir],
                      Length[Rest[reservoir]]];
display
```





Now, we could go on to make a three-dimensional (3D) crossword puzzle, crossword puzzles on a torus by identifying equivalent lattice points, and so on, but we end here to leave something for the reader.

Σ (* session summary *) **`TMGBs`PrintSessionSummary[]`**

6.5 Mathematical Operations with Matrices

■ 6.5.1 Linear Algebra

This section is devoted to problems arising in linear algebra. We include them in this chapter because of the identification *list* ≙ *vector*, *listOfLists OfEqualLength* ≙ *matrix*, etc., and because we need the corresponding operations later, especially in the next part of the book, which deals with graphics in *Mathematica*. (We will discuss a special topic from linear algebra in Chapter 1 of the Numerics volume [302✱] in connection with numerical methods, and in Chapter 1 of the Symbolics volume [303✱] when dealing with symbolic calculations; we will not touch this subject again.) Let us refer to the three functions `Dot`, `Inner`, and `Outer` discussed in Subsection 6.4.3. The following statement holds for nearly all commands from linear algebra. (*Mathematica* is a very useful tool for working with concrete matrices, for a collection of many useful matrix identities for symbolic matrices, see [192✱], [25✱].)

The commands used to solve problems in linear algebra (determinants, solution of systems of linear equations, eigenvalues, etc.) can in most cases be applied to arbitrary approximate numbers, exact numbers, and symbolic expressions if these operations are reasonably defined for these types of arguments. The runtime and the complexity depend dramatically on the form of the input.

Before operating on a matrix, it is useful to determine its structure and size. `Length` gives the information at level 1. To get the "size" of a matrix, we can use `Dimensions`.

`Dimensions[`*list*`]`
gives the dimensions of the matrix *list*. The head of *list* need not be `List`.

Thus we use `Dimensions` here.

```
Dimensions[Table[f[i, j, k, l],
                 {i, 3}, {j, 2, 3}, {k, 0, 3, 1/2}, {l, 0, 2}]]
```

For nonrectangular objects, the outermost dimension is found.

```
Dimensions[z[z[1], z[z[2], z[2]], z[z[z[3], z[3]], z[z[4], z[4]]]]]
```

We turn now to the typical problems of linear algebra: inverting a matrix, computing its determinant, calculating the eigenvalues and eigenvectors in $\mathbf{A}.\mathbf{x}_i = \lambda\, \mathbf{x}_i$, and solving systems of the form $\boldsymbol{A}.\boldsymbol{x} = \mathbf{b}$.

`Inverse[`*squareMatrix*`]`
finds the inverse matrix $squareMatrix^{-1}$ corresponding to the square matrix *squareMatrix*.

Here is the general definition of a Hilbert matrix.

```
hilbert[n_] := Table[1/(i + j + 1), {i, n}, {j, n}];
```

Here is a Hilbert matrix of order 6.

```
hilbert[6] // MatrixForm
```

`Dot[`*squareMatrix*`, Inverse[`*squareMatrix*`]]` gives the identity matrix (if *matrix* is not singular).

```
(%.Inverse[hilbert[6]]) // MatrixForm
```

A further important matrix operation is `Det` [221✱], [311✱], [85✱], [174✱], [175✱].

`Det[`*squareMatrix*`]`
finds the determinant of the matrix *squareMatrix*.

Here are the determinants of the first nine Hilbert matrices.

```
Table[Det[hilbert[n]], {n, 9}] // MatrixForm
```

At this point, we should make a remark about the resources required by the routines for linear algebra when applied to exact, approximate numerical, and symbolic arguments. We look at the computation times needed to find the determinant for several examples. Here are the numbers when the elements of the matrix are given with machine accuracy. (For more accurate timings, we repeat each calculation using the inner `Do` loop.)

```
Table[Timing[Do[Det[Array[N[1/Plus[##]]&, {n, n}]],
                {10}]][[1, 1]], {n, 10}]
```

This is what we get with 32-digit numbers as elements.

```
Table[Timing[Do[Det[Array[N[1/Plus[##], 32]&, {n, n}]],
               {10}]][[1, 1]], {n, 10}]
```

This is what we get with 512-digit numbers as elements.

```
Table[Timing[Do[Det[Array[N[1/Plus[##], 512]&, {n, n}]],
               {10}]][[1, 1]], {n, 10}]
```

Now, we use exact fractions as elements.

```
Table[Timing[Do[Det[Array[(1/Plus[##])&, {n, n}]],
             {10}]][[1, 1]], {n, 12}]
```

Finally, we get the following timings for symbolic arguments as elements (be aware that there is no inner `Do` loop in the following input).

```
Table[Timing[Det[Array[a, {n, n}]]; ][[1, 1]], {n, 7}]
```

The difference in the amount of time required is quite large. The calculation with approximate numbers with many digits or with symbolic quantities is much slower than is the computation with elements of machine accuracy. For matrix dimensions relevant to practical problems, the time required can differ by several orders of magnitude, and so the user should always think about when it is best to go to machine numbers. Finding determinants of dense matrices symbolically can also take a great deal of memory for $n \geq 8$.

```
Table[{dim, ByteCount[Det[Array[a, {dim, dim}]]]/10.^6 MB}, {dim, 8}]
```

The determinant $|\mathbf{1} - \mathbf{A}.\mathbf{B}|$ for antisymmetric matrices **A** and **B** can always be written as a square [319✱]. Here we show this explicitly for nonnumeric matrices of dimensions two to five. The complexity of the calculations grows quickly with the matrix dimensions.

```
(* antisymmetric d×d matrix with elements m[i, j] *)
AntisymmetricMatrix[d_, m_] :=
             Table[Which[j < i, m[i, j], i == j, 0, j > i, -m[j, i]],
                   {i, d}, {j, d}]

Module[{a}, Table[Timing[(Det[IdentityMatrix[d] -
              AntisymmetricMatrix[d, a].AntisymmetricMatrix[d, b]] //
              (* recognize a square *) Factor) /. _^2 -> aFullSquare],
      {d, 2, 5}]]
```

These remarks on computational time and memory requirements essentially apply to all linear algebra routines. Because *Mathematica* does not automatically make use of auxiliary variables to save intermediate expressions, this behavior is expected.

In the following example, we observe the order in which summands of the determinant with symbolic entries are computed. We compute the determinant of the square matrix f_{ij}, where with the elements we associate the rule that products of f are simplified to one f with the joining of the current arguments.

```
f[a__] f[b__] ^= f[a, b];
```

```
Det @ Array[f[{#1, #2}]&, {3, 3}]
```

```
Clear[f]
```

Now let us deal with some larger symbolic determinants. We will calculate the Wronskian [49✱] of the functions $\{\sin(z),\ \sin(2\,z),\ \ldots,\ \sin(n\,z)\}$. This defines the Wronskian $W_z(ws)$ of a list of functions ws with respect to the variable z.

```
Wronskian[ws_List, z_] := Det[Table[D[ws, {z, k}],
                                    {k, 0, Length[ws] - 1}]]
```

Here are the first eight Wronskians. They are relatively large sums.

```
Length /@
 (WSins = Table[Wronskian[Table[Sin[k z], {k, n}], z], {n, 8}])
```

```
Short[WSins[[8]], 6]
```

Simplifying the Wronskians using `TrigFactor` yields the short result $\left(\prod_{k=1}^{n-1} k!\right)\sin^n(z)\,(-2\sin(z))^{n\,(n-1)/2}$ [96✱], [335✱], [323✱], [290✱].

```
WSins // TrigFactor
```

```
Table[Product[k!, {k, n - 1}] Sin[z]^n (-2Sin[z])^(n (n - 1)/2), {n, 8}]
```

Another function often needed in matrix calculations is `Tr`.

`Tr[`*squareMatrix*`]`
  calculates the trace of *squareMatrix* (the sum of its diagonal elements).

As a side step, we will compare various top-level implementations of `Tr`. We can define such a function and even call it `Trace` as long as we keep it separate from the built-in `Trace` used for debugging. Of course, we could name it `MatrixTrace`, but partly the point of the following is the coexistence of two commands with the same name in different contexts. This coexistence can be done using *Mathematica*'s context specification. The built-in `Trace` is in the context `` System` ``. Thus, we need only define our `Trace` explicitly in the context `` Global` ``. Because the commands of the context `` Global` `` are applied before those in the context `` System` ``, we can make the following definition. (*Mathematica* warns us that we have now two functions called `Trace` in two contexts, which both are on the context path.)

```
SetAttributes[Global`Trace, HoldAll]
Global`Trace[x_?MatrixQ] := Sum[x[[i, i]], {i, Length[x]}]
Global`Trace[x_] := System`Trace[x]
```

Our `Trace` function is now available.

```
??Trace
```

It works for matrices.

```
Trace[{{1, 2}, {3, 4}}]
```

It also works for other expressions. (To get this, we needed the attribute `HoldAll`—otherwise, the argument would be computed before giving it to ``System`Trace``, and this would have led to no further computation.)

```
Trace[2 + 3 7 + 5 6 + Sin[Pi] + Log[E]]
```

Even the computation of the trace of a matrix can now be observed in detail using the built-in `Trace`.

```
Trace[Trace[{{1, 2}, {3, 4}}]]
```

Note that the above computation of the trace is, in a certain sense, the "obvious" one, but not necessarily the most elegant. Here are a few other implementations that do not make use of auxiliary variables (for comparison, we again give the "obvious" definitions).

```
traceDef1[mat_] := Plus @@ Transpose[mat, {1, 1}]

traceDef2[mat_] := Plus @@ Flatten[MapIndexed[Take, mat]]

traceDef3[mat_] := Sum[mat[[i, i]], {i, Length[mat]}]

traceDef4[mat_] :=
Plus @@ MapIndexed[#1[[#2[[1]]]]&, mat]

traceDef5[mat_] := Plus @@ First[Transpose[
MapIndexed[RotateLeft[#1, #2[[1]] - 1]&, mat]]]

traceDef6[mat_] := Plus @@
Flatten[IdentityMatrix[Length[mat]] mat]

traceDef7[mat_] :=
Fold[#1 + #2[[Position[mat, #2][[1, 1]]]]&, 0, mat]
```

(`traceDef6` follows [1✱].) Here is a test of their relative speeds for a 200×200 matrix. To get a reasonable resolution, we use an inner `Do` loop inside `Timing`.

```
testMatrix = Table[i j, {i, 200}, {j, 200}];
```

The built-in trace function `Tr` is of course the fastest.

```
Timing[Do[Tr[testMatrix], {10^5}]]
```

Here are the timings for our implementations. (Observe the different number of times the trace is carried out in the various examples.)

```
Timing[Do[traceDef1[testMatrix], {100}]]

Timing[Do[traceDef2[testMatrix], {100}]]

Timing[Do[traceDef3[testMatrix], {100}]]

Timing[Do[traceDef4[testMatrix], {100}]]

Timing[traceDef5[testMatrix]]

Timing[traceDef6[testMatrix]]

Timing[traceDef7[testMatrix]]
```

Next, we prove the identity $\partial \det(\mathbf{M}(\tau)) / \partial \tau = \det(\mathbf{M}(\tau)) \operatorname{tr}\left(\mathbf{M}'(\tau).\mathbf{M}(\tau)^{-1}\right)$ [246✱], [112✱] for a $n \times n$ matrix $\mathbf{M}(\tau)$ with τ-dependent matrix elements for small n. We use `Simplify` to show that the difference of the left-hand side and the right-hand side vanishes.

```
Module[{ℳ, τ},
      Table[ℳ = Table[𝓂[i, j][τ], {i, d}, {j, d}];
            zero = D[Det[ℳ], τ] - Det[ℳ] Tr[D[ℳ, τ].Inverse[ℳ]];
            {LeafCount[zero], Timing[Simplify[zero]]}, {d, 2, 4}]]
```

For a nonsingular matrix **A**, we have $\mathrm{Tr}(\mathbf{A}^{-1}) = \partial(\mathbf{A} - \lambda\,\mathbf{1})/\partial\lambda\,|_{\lambda=0}$ (here **1** is the identity matrix of the same dimension as **A**) [313✱]. The following input checks this identity for generic matrices of dimensions 1×1 to 6×6. Similar to the above calculation with symbolic matrices, the calculation times increase quickly with the dimension.

```
Module[{A, a},
Table[Timing[A = Table[a[i, j], {i, d}, {j, d}];
      ExpandAll[Tr[Inverse[A]] -
        D[Log[Det[A + λ IdentityMatrix[d]]], λ] /. λ -> 0]], {d, 6}]]
```

For 2×2 matrices $\mathcal{M}_k$ and $\mathbf{M}_k$, the $d\times d$ matrix **A** with elements $a_{i,j} = \mathrm{Tr}(\mathcal{M}_i.\mathbf{M}_j^{\pm 1})$ has the interesting property that its determinant vanishes identically if $d \geq 5$. Here is a quick explicit check for this unusual property for $d \leq 6$ [143✱].

```
ℳ[k_] = Table[𝓂[k][i, j], {i, 2}, {j, 2}];
M[k_] = Table[m[k][i, j], {i, 2}, {j, 2}];

Table[{d, Expand[Det @ Table[Tr[ℳ[k].M[l]],
                             {k, d}, {l, d}]] === 0},
      {d, 6}]

Table[{d, Expand[Det @ Table[Tr[ℳ[k].Inverse[M[l]]],
                             {k, d}, {l, d}]] === 0},
      {d, 6}]
```

To compute eigenvalues and eigenvectors, we have `Eigenvalues`. (Note that `values` and `system` in `Eigenval`⋱ `ues` and `Eigenvectors` are not capitalized.)

`Eigenvalues[`*squareMatrix*`]`
finds all eigenvalues of the matrix *squareMatrix*.

`Eigenvectors[`*squareMatrix*`]`
finds all eigenvectors of the matrix *squareMatrix*.

`Eigensystem[`*squareMatrix*`]`
finds all eigenvalues and eigenvectors of the matrix *squareMatrix*.

Presently, it is not possible to compute a selected set of eigenvalues and eigenfunctions (typically, we have large matrices but are only interested in the largest or smallest eigenvalue). Moreover, the built-in commands do not take into account the sparsity of matrices. In case of degenerate eigenvalues, the corresponding eigenvectors given by `Eigensys tem` or `Eigenvectors` spanning the eigenspace are not orthogonal to each other, but just linear independent. These eigenvectors can be easily orthogonalized (the function `GramSchmidt` from the package ``LinearAlgebra`Orthog onalization` `` comes in handy here.)

A measure of the (numerical) difficulty of finding the inverse of a symmetric matrix is given by its so-called condition number $|\max(\mathit{eigenvalue}) / \min(\mathit{eigenvalue})|$.

```
Do[ev = Eigenvalues[N[hilbert[i]]];
   Print["i = ", i, " condition number = ", Max[ev]/Min[ev]],
{i, 9}]
```

For comparison, we note that, for generalized eigenvalue problems from finite element method computations of dimension 50000, the condition number is typically in the order of magnitude of the last printed condition number.

The following (Pauli) matrices and their eigenvalues play an important role in the description of the inner rotational momentum (spin) of elementary particles (see any textbook on quantum mechanics, e.g., [321✱], [88✱], and [64✱]). (We represent σ_i as σ[*i*].)

```
σ[1] = {{0, 1}, {1, 0}};
σ[2] = {{0, -I}, {I, 0}};
σ[3] = {{1, 0}, {0, -1}};
Do[Print["σ[", i, "] =   ", MatrixForm[σ[i]]], {i, 3}]
```

These matrices have the following properties:

■ Their square is the identity matrix.

■ Their eigenvalues are $+1$ and -1.

■ $\sigma_i.\sigma_j = \sigma_k$ with i, j, k cyclic.

■ They are anticommutative, that is, $\sigma_i.\sigma_j = -\sigma_j\,\sigma_i$.

We quickly check these properties.

```
MatrixForm /@ Table[σ[i].σ[i], {i, 3}]
```

```
Table[Eigenvalues[σ[i]], {i, 3}]
```

```
{σ[1].σ[2] == I σ[3], σ[2].σ[3] == I σ[1], σ[3].σ[1] == I σ[2]}
```

```
{σ[1].σ[2] == -σ[2].σ[1], σ[2].σ[3] == -σ[3].σ[2], σ[3].σ[1] == -σ[1].σ[3]}
```

Here is the eigensystem for a spin in an arbitrary direction using direction cosines *d*[1], *d*[2], and *d*[3]. This eigensystem corresponds to the matrix $\sum_{i=1}^{3} \sigma_i\, d_i$.

```
Eigensystem[
Sum[d[i] σ[i], {i, 3}]] /. {d[1]^2 + d[2]^2 + d[3]^2 -> 1}
```

We have a more detailed look at a small symmetric matrix with real elements and nondegenerate eigenvalues, and quickly review some of the properties of the eigenvalues and eigenvectors. We will use a matrix $\mathcal{M}$ with elements $m_{ij} = i\,j \big/ \left(i^2 + j^2 + 1\right)$.

```
symmMatrix[n_] := Table[i j/(1 + i^2 + j^2), {i, n}, {j, n}]
```

For the explicit calculations, we will use a 6×6 matrix $\mathcal{M}$ (meaning $n = 6$).

```
(M = symmMatrix[6]) // MatrixForm
```

These are the eigenvalues ω_j and the eigenvectors $\boldsymbol{e}_j$.

```
{evals, evecs} = Eigensystem[N[M, 10]]
```

The eigenvectors to different eigenvalues are orthogonal to each other and the eigenvectors are normalized.

```
Table[evecs[[j]].evecs[[k]], {j, Length[evecs]}, {k, Length[evecs]}]
```

Using the outer product, we can form the eigenprojectors $\mathcal{E}_j = \boldsymbol{e}_j \otimes \boldsymbol{e}_j$. The eigenprojectors project on the subspace that is spanned by all vectors $\boldsymbol{g}_j$, such that $\mathcal{M}.\boldsymbol{g}_j = \omega_j\,\boldsymbol{g}_j$.

```
Es1 = Table[Outer[Times, evecs[[j]], evecs[[j]]], {j, Length[evals]}];
```

The eigenprojectors can be expressed as a matrix product that figures only the eigenvalues:

$$\mathcal{E}_j = \prod_{\substack{k=1 \\ k\neq j}}^{n} \frac{1}{\omega_j - \omega_k} (\mathcal{M} - \omega_k \mathbb{1}_n)$$

```
ℰs2 =
With[{n = Length[evals]},
      Table[Fold[Dot, IdentityMatrix[n], #]& @ (* the factors *)
       Table[If[j == k, IdentityMatrix[n], 1/(evals[[j]] - evals[[k]])*
               (symmMatrix[n] -  evals[[k]] IdentityMatrix[n])], {k, n}],
      {j, n}]];
```

The two sets of eigenprojectors are identical within the precision of the the calculation.

```
ℰs1 - ℰs2 // Flatten // N[#, {Infinity, 1}]& // Union
```

The eigenprojectors are symmetric matrices.

```
Max[# - Transpose[#]]& /@ ℰs1
```

The eigenvalues of projection matrices are 0 and 1.

```
(Eigenvalues /@  ℰs1) /. _?(Abs[#] < 10^-20&) :> 0
```

The differences of the projectors to the identity operator are also projectors.

```
(Eigenvalues /@  ((IdentityMatrix[6] - #)& /@ ℰs1)) /.
                                  _?(Abs[#] < 10^-20&) :> 0
```

This means their trace is 1 too.

```
Tr /@ ℰs1
```

The eigenvectors lie within the eigenspaces of the eigenprojectors.

```
Table[ℰs1[[j]].evecs[[j]] - evecs[[j]], {j, 6}]
```

The sum of all eigenprojectors is the identity matrix (meaning the eigenvectors span the whole space) and the sum of the products of the eigenprojectors (the spectral resolution) with the eigenvalues is the original matrix $\mathcal{M} = \sum_{j=1}^{n} \omega_j \mathcal{E}_j$.

```
(Plus @@ (ℰs1))
```

```
(Plus @@ (evals ℰs2)) - ℳ
```

The eigenprojectors are themselves orthogonal to each other. But because the eigenprojectors are themselves matrices, we must now sum over two indices.

```
Table[(* form trace of dot product *) Tr[ℰs1[[j]].ℰs1[[k]]],
      {j, Length[evals]}, {k, Length[evals]}]
```

`Eigensystem` works for dense numerical matrices up to around $10^3 \times 10^3$ (depending on the computer used, this number might be too small or too large) in a few minutes. Here is a nonsymmetric 20×20 tridiagonal matrix.

```
hm = Table[Which[i == j, 5, j - i == 1, i + j,
                 i - j == 1, i + j + 1, True, 0], {i, 20}, {j, 20}];
```

Here is a submatrix.

```
TableForm[Take[#, 8]& /@ Take[hm, 8]]
```

This example gives its eigenvalues (first list of the following output) and eigenvectors (second list).

```
({evals, evecs} = Eigensystem[N[hm]]) // Short[#, 12]&
```

We can verify the correctness of this result by checking the equation $\mathbf{A}_{diag} = \mathbf{C}^{-1}.\mathbf{A}.\mathbf{C}$, where **C** is the matrix whose columns are the eigenvectors and $\mathbf{A}_{diag}$ is the diagonal matrix with the eigenvalues of **A** on the main diagonal. Because the columns of **C** are the eigenvectors, we first have to transpose `evecs`. We set insignificant components ($< 10^{-10}$) to 0.

```
chop[m_] := m //. _?(Abs[#] < 10^-10&) -> 0

Inverse[Transpose[evecs]].hm.Transpose[evecs] // chop
```

Here are the same eigenvalues as computed with `Eigensystem`.

```
Union @@ (DiagonalMatrix[evals] - % // chop)
```

The standard deviation of the eigenvalues (in case of real eigenvalues) can be expressed through the traces of the matrix and its square [114✱], [328✱].

```
With[{n = Length[evals]},
     {(* direct evaluation *) (n - 1)/n Variance[evals],
      (* through traces *) 1./n (Tr[hm.hm] - Tr[hm]^2/n)}]
```

For matrices consisting of exact numbers, we can find the eigenvalues and eigenfunctions when the characteristic equation can be solved exactly in radicals, as well as for higher order characteristic equations, which means that typically matrices of at most 4×4 can be treated symbolically if we only want at most radicals in the result. For larger matrices, the eigenvalues will (in most cases unavoidably) be expressed in `Root`-objects; see Chapter 1 of the Symbolics volume [303✱] for a detailed discussion of them.

```
Eigenvalues[hilbert[6]]
```

Of course, numerically, no problem exists in calculating the eigenvalues.

```
Eigenvalues[N[hilbert[6]]]
```

Also, eigenvalues can be calculated in arbitrary precision.

```
Eigenvalues[N[hilbert[6], 50]]
```

Eigenvalue problems are very important in practical applications. One way to calculate eigenvalues iteratively is the so-called power method for calculating the lowest eigenvalue. With *Mathematica*, we can implement this method very concisely (see, for instance, [324✱], [316✱], [299✱], [92✱], [128✱], and [91✱]). Here is the lowest eigenvalue of an example matrix calculated with this method.

```
Union[Function[matrix, matrix.Last[#]/Last[#]&[
 FixedPointList[N[(#/Max[#])&[matrix.#]]&, {1, 1, 1, 1}, 100]]][
 (* the matrix *)
 {{1, -3, 2, -4}, {4, -4, 1, 3}, {6, 3, -5, 6}, {3, -5, 5, -6}}],
                SameTest -> Equal]
```

Here is the comparison with the direct result of *Mathematica* (the construction `HeldPart[…]` does the extraction of the matrix used in the last computation for the input history and saves us from retyping the matrix.).

```
Eigenvalues[N @ HeldPart[(Hold /@ DownValues[In][[
                      $Line - 1]])[[2]], 1, 1, 1][[1]]]
```

The following example calculates the integer-valued eigenvalues of a complicated-looking matrix [52✱], [53✱], [54✱], [166✱].

```
neatMatrix[n_] :=
  Table[I If[j === k, Sum[If[i === j, 0, Cot[j - i]], {i, n}],
            1/Sin[j - k]], {j, n}, {k, n}]

neatMatrix[4]
```

```
Eigenvalues[N[%]]
```

```
Eigenvalues[N[neatMatrix[30]]]
```

```
Sort[Re[%]]
```

For other matrices that have nice eigenvalues, see [250✱], [31✱].

Be aware of the imaginary parts in the last eigenvalue result. Although `neatMatrix[30]` was explicitly hermitian the eigenvalues returned were not purely real. The imaginary parts resulted from the algorithm used in *Mathematica*. Using a higher precision of the input matrix results in smaller imaginary parts.

```
Eigenvalues[N[neatMatrix[30], $MachinePrecision + 1]] // Im // N
```

Interestingly, for every even n, the eigenvalues of `neatMatrix` are $(-n+1)$, $(-n+3)$, ..., -1, +1, …, $(n-3)$, $(n-1)$, [52✱].

```
Sort[Re[Eigenvalues[N[neatMatrix[100]]]]] // Timing
```

The eigenvalues returned by `Eigenvalues` and `Eigensystem` are sorted by absolute value of the real part. The following graphics show the size of the eigenvalues and a density plot of the eigenvectors of a 400×400 matrix with elements $a_{i\,j} = \tan(7/9\,(i+j))$.

```
efGraphics[f_, dim_] :=
Module[{mat, evals, evecs},
 (* the matrix *)
 mat = Table[N[f[i, j]], {i, dim}, {j, dim}];
 (* the eigenvalues and eigenvectors *)
 {evals, evecs} = Eigensystem[mat];
 (* the eigenvalues and eigenvectors *)
 Show[GraphicsArray[{
 ListPlot[Sort[Re[evals]], PlotJoined -> True, PlotRange -> All,
          Axes -> False, Frame -> True, DisplayFunction -> Identity],
 (* sort eigenvectors *)
 evecsSorted = evecs[[First /@ Sort[
      MapIndexed[{#2[[1]], #1}&, Re[evals]], #1[[2]] < #2[[2]]&]]];
 (* density plot of eigenvectors *)
 ListDensityPlot[Re[evecsSorted], Mesh -> False, FrameTicks -> None,
                 ColorFunction -> (Hue[0.8 #]&),
                 DisplayFunction -> Identity]}]]]
```

```
efGraphics[Tan[7/9(#1 + #2)]&, 400]
```

Next, we display the eigenvalues of 16×16 matrices with elements

$$a_{i\,j} = \begin{cases} \exp(i\,\varphi) & \text{if } i < j \\ \exp(-i\,\xi\,\varphi) & \text{if } i > j \\ 1 & \text{if } i = j \end{cases}$$

as φ ranges from 0 to 2π and ξ is a fixed constant. The eigenvalues for each value of φ are displayed as points of the same color in the complex plane.

```
Show[GraphicsArray[
Function[ξ, With[{d = 16, ppφ = 2400},
Graphics[{PointSize[0.001],
Table[{Hue[φ/(2Pi)], Point[{Re[#], Im[#]}]& /@ Eigenvalues[
        (* the d×d matrix *)
        Table[Exp[I φ Which[i < j, 1., i > j, -1. ξ, i == j, 0.]],
              {i, d}, {j, d}]]},
     {φ, 0, 2Pi, 2Pi/ppφ}]}, PlotRange -> {{-2, 4}, {-3, 3}},
     AspectRatio -> Automatic]]] /@ (* ξ-values *) {1/6, 2, 3, 5}]]
```

Modifying the definition of the last matrix slightly, we get a much more complicated pattern of the eigenvalues.

```
Module[{n = 100, pp = 400, A, φ},
(* define a matrix M[φ] *)
Set @@ {A[φ_], Table[
          Which[Abs[i - j] === 1, Exp[I 2Pi Sign[i - j] j φ],
                Abs[i - j] === 3, 1, True, 0], {i, n}, {j, n}]};
(* show real and imaginary parts of eigenvalues as a function of φ *)
Show[GraphicsArray[{# /. Point[{x_, y_}] :> Point[{Re[x], y}],
                    # /. Point[{x_, y_}] :> Point[{Im[x], y}]}&[
Graphics[{PointSize[0.002], {#, Apply[{#1, 1 - #2}&, #, {-2}]}&[
     Table[Point[{#, φ}]& /@ Sort[Eigenvalues[A[φ]]],
           {φ, 0., 1., 1./pp}]]},
    Frame -> True, PlotRange -> All, FrameTicks -> None]]]]]
```

Let us give a small graphics application of `Eigensystem`: The use of the eigenmesh to smooth a curve [115✱]. Given a curve with points $\{x_k, y_k\}$ we express the curve as a superposition of the eigenfunctions of a finite difference approximation of the curvature. The following input uses a noisy Lissajous curve with 512 points. The graphic shows how the curve is reproduced when all eigenfunctions are taken into account.

```
Module[{n = 512, mat, evals, evecs, xData, yData,
        scpsx, scpsy, sumx, sumy},
(* matrix of the Laplace operator *)
mat = Table[Which[i === j, 1, Abs[i - j] === 1, -1/2,
            (i == 1 && j == n) || (i == n && j == 1),
            -1/2, True, 0], {i, n}, {j, n}] // N;
(* eigensystem of mat *)
{evals, evecs} = Eigensystem[mat];
(* sort eigenvectors *)
evecs1 = evecs[[First /@ Sort[
     MapIndexed[{#2[[1]], #1}&, Re[evals]], #1[[2]] < #2[[2]]&]]];
{xData, yData} = Transpose[
          Table[{Cos[5. t] + 6/5 Random[], Sin[3. t] + 6/5 Random[]},
                {t, 0, 2Pi, 2Pi/(n - 1)}]];
{scpsx, scpsy} = {xData.#& /@ #, yData.#& /@ #}&[evecs1];
sumx = 0; sumy = 0;
Show[Graphics[Reverse @
Table[{sumx, sumy} = {sumx, sumy} +
                     {scpsx[[k]] evecs1[[k]], scpsy[[k]] evecs1[[k]]};
      {Hue[k/n 0.8], Line[Transpose[{sumx, sumy}]]},
      {k, n}]], AspectRatio -> Automatic]]
```

While for many applications, symmetric (hermitian) matrices are most important, asymmetric matrices can have quite interesting properties too. In the following, we will visualize the (generically complex) eigenvalues of 32768 matrices of size 16×16. The elements of the matrices are all 1 on the upper subdiagonal ($a_{i,i+1} = 1$), and all permutations of ± 1 on the lower subdiagonal ($a_{i,i-1} = \pm 1$) [140✱]. `makeTridiagonalMatrix` constructs such a matrix for a given subdiagonal *subDiagonal*.

```
permutationsPM1[d_] := permutationsPM1[d] = Flatten[
 Permutations /@ Table[Join[Table[1, {k}], Table[-1, {d - k}]],
                       {k, 0, d}], 1];

makeTridiagonalMatrix[subDiagonal_] :=
Module[{d = Length[subDiagonal] + 1, M},
       (* matrix to be filled *)
       M = Table[0., {d}, {d}];
       (* add 1's *)
       Do[M[[i, i + 1]] = 1., {i, d - 1}];
       (* add ±1's *)
       Do[M[[i, i - 1]] = N[subDiagonal[[i - 1]]], {i, 2, d}];
       (* return matrix *)  M]
```

The resulting 524288 eigenvalues form a complicated pattern in the complex plane. (Using larger matrices constructed in the same way shows the fractal nature of the resulting point set [140✱].)

```
Show[Graphics[{PointSize[0.001],
     ((* make points in the complex plane *)
      Point[{Re[#], Im[#]}]& /@
        Eigenvalues[makeTridiagonalMatrix[#]])& /@
                                    permutationsPM1[15]}],
     AspectRatio -> Automatic, PlotRange -> All]
```

Here is another tridiagonal matrix. Its determinant is the polynomial $x^n + \sum_{k=0}^{n-1} c_k\, x^k$ [89✱].

```
tridiagonalPolyMatrix[x_, c_List?(OddQ[Length[#]]&)] :=
With[{o = Length[c]}, (-1)^((o - 1)/2) *
Table[Which[i == j, If[OddQ[i], c[[o - i + 1]] +
                       If[i == 1, x, c[[o - i + 2]] x], 0],
            j == i - 1, If[OddQ[i], x, -1],
            j == i + 1, If[OddQ[i], -1, x],
            True, 0], {i, o}, {j, o}]]

tridiagonalPolyMatrix[x, {α, β, γ, δ, ϵ}] // Det
```

The next two graphics show the eigenvalues of the matrix when x and the list c are varied.

```
Show[GraphicsArray[
Block[{$DisplayFunction = Identity, o = 200, p = 60},
{(* vary c values *)
 Graphics[{PointSize[0.004],
 Table[{Hue[0.78 ρ/2], Point[{Re[#], Im[#]}]& /@
   Eigenvalues[tridiagonalPolyMatrix[1 - I,
                N @ Table[ρ Exp[2 Pi I k/o], {k, 0, o}]] ]},
      {ρ, 0, 2, 2/p}]}, PlotRange -> All],
 (* vary x value *)
 Graphics[{PointSize[0.004],
 Table[{Hue[0.78 ρ], Point[{Re[#], Im[#]}]& /@
   Eigenvalues[tridiagonalPolyMatrix[Exp[2 Pi I ρ],
                N @ Table[Exp[2 Pi I k/o], {k, 0, o}]] ]},
      {ρ, 0, 1, 1/p}]}, PlotRange -> All]}]]]
```

For demonstration purposes, we will diagonalize the following symbolic 2×2 matrix. Assuming the three parameters α, β, and γ are real-valued, this is a general hermitian 2×2 matrix.

```
ℋ = {{α, γ + I δ}, {γ - I δ, β}};
```

To complex conjugate symbolic quantities, we now introduce the function conjugate that carries out the complex conjugation for all complex numbers.

```
conjugate[expr_] := expr /. Complex[r_, i_] :> Complex[r, -i]
```

It is straightforward to calculate the diagonalizing matrix. As typical in the context of hermitian matrices, we call it **U**.

```
(* find eigenvalues and eigenvectors *)
{eigenValues𝓗, eigenVectors𝓗} = Eigensystem[𝓗];

(* a quick check for the eigensystem *)
Table[𝓗.eigenVectors𝓗[[j]] -
      eigenValues𝓗[[j]] eigenVectors𝓗[[j]], {j, 2}] // Simplify

(* norms of the eigenvectors *)
norms𝓗 = Sqrt[#.conjugate[#]]& /@ eigenVectors𝓗 // Simplify;
(* normalize eigenvectors *)
eigenVectors𝓗N = Divide @@@ Transpose[{eigenVectors𝓗, norms𝓗}] //
                 Simplify;
(* the eigenvectors as columns are the diagonalizing matrix *)
U𝓗 = Transpose[eigenVectors𝓗N]
```

U is unitary and fulfills the properties $\mathbf{U}.\overline{\mathbf{U}}^T = \mathbf{1}$ and $\overline{\mathbf{U}}^T.\mathbf{U} = \mathbf{1}$.

```
U𝓗Adjoint = conjugate[Transpose[U𝓗]];

{U𝓗.U𝓗Adjoint, U𝓗Adjoint.U𝓗} // Simplify
```

And the original matrix $\mathcal{H}$ can be expressed as $\mathbf{U}.\boldsymbol{\mathcal{E}}.\overline{\mathbf{U}}^T = \boldsymbol{\mathcal{H}}$ where $\boldsymbol{\mathcal{E}}$ is the diagonal matrix of the eigenvalues.

```
U𝓗.DiagonalMatrix[eigenValues𝓗].U𝓗Adjoint - 𝓗 // Simplify
```

Sometimes one has to solve the so-called generalized eigenvalue problem $\mathbf{A}.\boldsymbol{x} = \lambda\,\mathbf{M}.\boldsymbol{x}$ with two square matrices **A** and **M**. For matrices with machine number elements, the form `Eignesystem[{`*A*`, `*M*`}]`, `Eigenvalues[{`*A*`, `*M*`}]`, and `Eigenvectors[{`*A*`, `*M*`}]` can be used for this case. Here is a small example.

```
Agep = {{1, 3}, {-2, 1}} // N;
Mgep = {{2, 3}, {9, -8}} // N;

{{eval1, evec1}, {eval2, evec2}} = Transpose @ Eigensystem[{Agep, Mgep}]
```

And here is a quick check of the numerical correctness of the result.

```
{Agep.evec1 - eval1 Mgep.evec1, Agep.evec2 - eval2 Mgep.evec2}
```

To solve systems of linear equations, we have `LinearSolve`.

`LinearSolve[`*matrix*`, `*rightHandSide*`]`

finds *x* so that *matrix.x* = *rightHandSide*. If the system of equations is underdetermined, `LinearSolve` gives one possible solution.

Here is a system that is clearly underdetermined because twice as many variables exist as equations.

```
m = 6;
(mat = Table[If[i <= 2j, 1, 0], {j, m}, {i, 2 m}]) // TableForm
```

We find the right-hand side.

```
rightHandSide = Table[i, {i, m}]
```

`LinearSolve` gives one possible solution.

```
LinearSolve[mat, rightHandSide]
```

If we want to find all solutions of a system of equations, we need `Solve`.

`Solve[{`*equations*`}, {`*unknowns*`}]`

solves the system of equations *equations* for the variables *unknowns*. If the system is underdetermined, some of the variables in the list *unknowns* are expressed in terms of other variables. The equations appearing in *equations* must have the head `Equal`.

Consider the following list of unknowns.

```
Clear[x]
unknowns = Table[x[i], {i, 2 m}]
```

We form the matrix product of `mat` with the coefficient vector `unknowns`.

```
leftHandSide = unknowns.#& /@ mat
```

With `Thread`, we can join the sides of the equations that belong together with `Equal` (standing for "equality in the mathematical sense").

```
(equations =
 Thread[Equal[leftHandSide, rightHandSide]]) // TableForm
```

`Solve` does what we expect: The complete solution of this underdetermined system depends parametrically on six variables.

```
Solve[equations, unknowns]
```

The result of `Solve` is a list of lists. The inner `List` contains the solution in form that uses `Rule`, so that it is easy to plug this solution into an expression. Here is another example for an underdetermined system. Given two vectors `{ax, ay, az}` and `{bx, by, bz}`, we look for a third `{𝓍, 𝓎, 𝓏}`, which is orthogonal to the two given ones [107✱].

```
Solve[{ax 𝓍 + ay 𝓎 + az 𝓏 == 0, bx 𝓍 + by 𝓎 + bz 𝓏 == 0}, {𝓍, 𝓎, 𝓏}]
```

```
({𝓍, 𝓎, 𝓏} /. %)[[1]]
```

```
{%.{ax, ay, az}, %.{bx, by, bz}} // Simplify
```

If the coefficients appearing in linear equations are floating point numbers, then we can tackle much larger problems using `Solve`. We consider a discretization of the functional equation

$$x(t) = \mathbb{f}(t,\ x(t) - \lfloor x(t) \rfloor)$$

$$\mathbb{f}(t, x) = \begin{cases} -\frac{x}{4} & \text{if } 0 \leq t < \frac{1}{4} \\ -\frac{1}{4} + \frac{3x}{4} & \text{if } \frac{1}{4} \leq t < \frac{1}{2} \\ \frac{1}{2} - \frac{x}{4} & \text{if } \frac{1}{2} \leq t < \frac{3}{4} \\ \frac{1}{4} + \frac{3x}{4} & \text{if } \frac{3}{4} \leq t < 1 \end{cases}$$

at $t_k = k / \nu$ for $\nu = 10^4$. `eqs` is a list of 10^4 linear equations for the $x_k = x(t_k)$. We explicitly insert `1.` to obtain numerical coefficients.

```
ν = 10000;
eqs = Table[1. x[t] - 1. Function[{t, x},
         Which[0   <= t < 1/4, -x/4,
               1/4 <= t < 1/2, -1/4 + 3x/4,
               1/2 <= t < 3/4, 1/2 - x/4,
               3/4 <= t <   1, 1/4 + 3/4 x]][
                                 t, x[FractionalPart[4t]]],
           {t, 0, 1 - 1/ν, 1/ν}];
```

Solving the 10^4 equations can be done in less than a minute on a 2 GHz computer.

```
(sol = Solve[# == 0& /@ eqs,
             Table[x[t], {t, 0, 1 - 1/ν, 1/ν}]]); // Timing
```

Displaying the connected points $\{x_k, 1 - x_{\nu-k}\}$ yields the so-called Siamese sisters [77✱].

```
Show[Graphics[{Thickness[0.001],
               MapIndexed[{Hue[#2[[1]]/ν], Line[#1]}&,
      Partition[Table[{x[t], 1 - x[1 - t]},
                      {t, 0, 1 - 1/ν, 1/ν}] /.
           x[1] -> x[0] /. Dispatch[sol[[1]]], 2, 1]]}],
      AspectRatio -> Automatic]
```

Because the "length" of the new vector is undetermined, we do not get a unique result.

With these matrix operations, we can easily do some calculations on the electric and magnetic field strengths `E` and `H` in a moving coordinate system.

Physical Remark: Lorentz Transformation of Physical Quantities

In the framework of the theory of special relativity, space and time coordinates are combined into one quantity $x_\mu = (x, y, z, ict)$. It is today common to use covariant and contravariant quantities (see, e.g., [222✱], [135✱], [255✱], [336✱], and [220✱]) instead of explicit vectors containing $i = \sqrt{-1}$, but here the use of a particular coordinate system is more convenient. (We discuss covariant and contravariant quantities in Chapter 1 of the Symbolics volume [303✱].) If we change from a coordinate system K to a system K', which is moving with constant relative velocity v along the x axis, space and time are transformed according to $x'_\mu = L_{\mu\nu}\, x_\nu$, where the expression with double subscripts is summed over 1 to 4, which in this case means over ν.

The matrix **L** (Lorentz transformation) is

$$\begin{pmatrix} (1-\beta^2)^{-1/2} & 0 & 0 & i\,\beta\,(1-\beta^2)^{-1/2} \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -i\,\beta\,(1-\beta^2)^{-1/2} & 0 & 0 & (1-\beta^2)^{-1/2} \end{pmatrix}$$

with $\beta = v\,/\,c$ (c = speed of light in a vacuum).

The quantities $\mathbf{E}(= E_k)$ and $\mathbf{H}(= H_k)$ appearing in the Maxwell equations (in a vacuum) can be combined similarly into a new quantity, the electromagnetic field strength tensor **F** ($F_{\mu\nu}$).

$$\begin{pmatrix} 0 & H_z & -H_y & -i\,E_x \\ -H_z & 0 & H_x & -i\,E_y \\ H_y & -H_x & 0 & -i\,E_z \\ i\,E_x & i\,E_y & i\,E_z & 0 \end{pmatrix}$$

As a tensor, its transformation is given by $F'_{\mu\nu} = L_{\mu\alpha}\, L_{\nu\beta}\, F_{\alpha\beta}$. (For more details, see any textbook on electrodynamics, e.g., [29✱] and [237✱]; for a three-dimensional tensor formulation, see [120✱].)

Now, we want to use these equations to find the electric and magnetic field strengths in a moving coordinate system. Here is the Lorentz transformation matrix.

```
Clear[v, c, β, x, y, z, t];
β = v/c;
LorentzTrafo = {{1/Sqrt[1 - β^2],     0, 0, I β/Sqrt[1 - β^2]},
                {0,                      1, 0, 0              },
                {0,                      0, 1, 0              },
                {-I β /Sqrt[1 - β^2], 0, 0, 1/Sqrt[1 - β^2]  }};
```

Its determinant is 1, which means that absolute space-time volumes are not altered by the transformation of coordinates.

```
Det[LorentzTrafo] // Simplify
```

Here is the four-vector of space-time.

```
fourX = {x, y, z, I c t};
```

Now, space and time are transformed from `fourX` to `fourXs` as follows. (Note that we must "dimensionalize" `fourXs` before the computation of the transformed quantities: `fourXs[1]` can be given a value, but not `fourXs[[1]]`; `fourXs = {, , , }` would indeed suffice, but it is visually ugly and generates messages.)

```
fourXs = {Null, Null, Null, Null};
Do[fourXs[[i]] = Sum[LorentzTrafo[[i, j]] fourX[[j]], {j, 4}], {i, 4}];
fourXs // Simplify
```

We can get the same result (faster) with matrix multiplication.

```
LorentzTrafo.fourX // Simplify
```

(We could also have used matrix multiplication for the above summation purpose.) The time coordinate $x_4/(ic)$ can be written in a more elegant form.

```
fourXs[[4]]/(I c) // Simplify
```

In the limiting case $c \to \infty$, we get exactly $x' = x - v\,t$ and $t' = t$, that is, the Galilean transformation. Here is the field strength tensor.

```
F = {{    0,    Hz,  - Hy, -I Ex},
     { - Hz,     0,    Hx, -I Ey},
     {   Hy,  - Hx,     0, -I Ez},
     { I Ex,  I Ey,  I Ez,     0}};
```

Now, we extract the electric and magnetic field strengths.

```
electricFieldStrength[fieldTensor_] :=
{fieldTensor[[4, 1]], fieldTensor[[4, 2]], fieldTensor[[4, 3]]}/I;

magneticFieldStrength[fieldTensor_] :=
{fieldTensor[[2, 3]], fieldTensor[[3, 1]], fieldTensor[[1, 2]]};
```

In the original coordinate system, we get exactly **E** and **H**.

```
electricFieldStrength[F]

magneticFieldStrength[F]
```

With the approach above, we get the field strength tensor in the new (moving) coordinate system.

```
FTrafo = Table[
  Sum[LorentzTrafo[[i, k]] LorentzTrafo[[j, l]] F[[k, l]],
      {k, 4}, {l, 4}], {i, 4}, {j, 4}] // Simplify
```

Again, by matrix multiplication, we can arrive at the same result.

```
LorentzTrafo.F.Transpose[LorentzTrafo] // Simplify
```

Thus, we obtain the "new" electric and magnetic field strengths.

```
Es = electricFieldStrength[FTrafo]

Hs = magneticFieldStrength[FTrafo]
```

Although both **E** and **H** vary, quantities that remain constant under a transformation of variables exist: $\mathbf{E}^2 - \mathbf{H}^2$ and $\mathbf{E}.\mathbf{H}$.

```
Es.Es - Hs.Hs // Simplify

Es.Hs // Simplify
```

These two invariants [103✱] can also be found from the field strength tensor and the so-called dual field strength tensor F^*, defined by

$$F^*_{\mu\nu} = \epsilon_{\mu\nu\alpha\beta}\, F_{\alpha\beta}$$

where $\epsilon_{\mu\nu\alpha\beta}$ is the complete antisymmetric tensor of fourth order (the Levi–Civita tensor, discussed earlier in Subsection 6.1.2).

```
LeviCivitaε[var__] := Signature[{var}]

FDual = Table[Sum[LeviCivitaε[i, j, k, l] F[[k, l]],
                  {k, 4}, {l, 4}], {i, 4}, {j, 4}];
MatrixForm[FDual]
```

Using matrix operations, we can carry out the last operation without explicitly using iterators.

```
LeviCivitaε4D = Table[LeviCivitaε[k, l, i, j],
                      {k, 4}, {l, 4}, {i, 4}, {j, 4}];

Tr[Transpose[LeviCivitaε4D.F, {3, 1, 4, 2}], Plus, 2] // MatrixForm
```

Now, we can express the invariants in the following way: $-2\left(\mathbf{E}^2 - \mathbf{H}^2\right)$.

```
Sum[F[[i, j]] F[[i, j]], {i, 4}, {j, 4}]
```

Again, by using matrix operations a short and efficient method of calculating the last result is obtained.

```
-Tr[F.F]
```

We also get the second invariant (up to the numerical factor of $-8\,i$).

```
Sum[FDual[[i, j]] F[[i, j]], {i, 4}, {j, 4}]
```

The matrix form of the last input is similar to the one from above.

```
-Tr[FDual.F]
```

Also the eigenvalues of $F_{\alpha\beta}$ can be expressed through the two invariants $\mathbf{E}^2 - \mathbf{H}^2$ and $\mathbf{E}.\mathbf{H}$.

```
(Eigenvalues[F] // Simplify) //.
          {Ex^2 + Ey^2 + Ez^2 - Hx^2 - Hy^2 - Hz^2 -> -C1,
           Ex Hx + Ey Hy + Ez Hz -> C2}
```

This result concludes our little detour into classical electrodynamics, but many other things could now be studied, for example, how to move relative to a given electromagnetic field to observe it as only an electric field or only as a magnetic field, and so on; we come back to this subject in Chapters 1 and 2 of the Graphics volume [301✱].

In many applications, the following situation occurs. We have more equations than unknowns, and all the equations together have to be fulfilled "as well as possible". The tool (in the linear case) for achieving this result is the function `PseudoInverse` [33✱], [284✱].

```
?PseudoInverse
```

PseudoInverse[*matrix*]
gives the Moore–Penrose inverse of *matrix*.

The Moore–Penrose inverse of a matrix $\tilde{\mathbf{A}}$ of a matrix **A** is uniquely defined by the following four properties:

- $\mathbf{A}.\tilde{\mathbf{A}}.\mathbf{A} = \mathbf{A}$
- $\tilde{\mathbf{A}}.\mathbf{A}.\tilde{\mathbf{A}}. = \tilde{\mathbf{A}}$
- $\left(\mathbf{A}.\tilde{\mathbf{A}}\right)^{T} = \mathbf{A}.\tilde{\mathbf{A}}$
- $\left(\tilde{\mathbf{A}}.\mathbf{A}\right)^{T} = \tilde{\mathbf{A}}.\mathbf{A}$

As an application of the PseudoInverse, let us calculate the "best" approximation of an intersection of a bunch of lines that nearly intersect in one point.

The implicit equation of a line going through a given point {p0x, p0y} with a given direction {dx, dy} (we discuss Eliminate in Chapter 1 of the Symbolics volume [303✱]) is given by the following expression.

```
Subtract @@ Eliminate[
   Thread[{x, y} == {p0x, p0y} + t {dx, dy}], {t}] // Simplify
```

Here are 12 lines with random slopes, all going "nearly" through the point {1/2, 1/2}. We represent these lines in the form {*point*, *direction*}.

```
tab = Table[{1/2 + {Abs[Sin[k]], Abs[Cos[k]]}/10.,
             1.{Sin[k E], Cos[k GoldenRatio]}}, {k, 12}]
```

Here is a sketch of the situation at hand.

```
Show[Graphics[{Line[{#[[1]] - 200 #[[2]],
                     #[[1]] + 200 #[[2]]}]& /@ tab}],
     PlotRange -> {{0, 1}, {0, 1}}, Frame -> True,
     AspectRatio -> Automatic]
```

res contains the implicit equations of the 12 lines.

```
res = #[[2, 2]] (#[[1, 1]] - x) + #[[2, 1]] (y - #[[1, 2]])& /@ tab
```

Let us look for the "point" of intersection. The best we can do is to solve the above system as well as possible in the sense to keep all squared differences minimal. Here is the brute force approach.

```
sol = Solve[{D[#, x] == 0, D[#, y] == 0}&[
            Expand[Plus @@ (res^2)]], {x, y}]
```

Here is the Moore–Penrose approach. We make a list of the parameters of the lines.

```
lineData = Module[{cx, cy, constant},
                     (* could use CoefficientList from
           Chapter 1 of the Symbolics volume here *)
                     cx = Cases[#, _ x][[1]]/x;
                     cy = Cases[#, _ y][[1]]/y;
                     constant = # - cx x - cy y // Chop;
                     {cx, cy, constant}]& /@ Expand[res]
```

This is the matrix constructed.

```
A = Take[#, 2]& /@ lineData;
```

Here is right-hand side.

```
b = Last /@ lineData;
```

We arrive at the same coordinates for the “best” crossing point.

```
PseudoInverse[A].b
```

Here the calculated point (as the center of the concentric circles) and the lines are shown.

```
Show[Graphics[{{GrayLevel[1/2],
                Table[Circle[{x, y} /. sol[[1]], r], {r, 0, 0.1, 0.01}]},
                Line[{#[[1]] - 200 #[[2]], #[[1]] + 200 #[[2]]}]& /@ tab}],
     PlotRange -> {{0.3, 0.7}, {0.3, 0.7}},
     AspectRatio -> Automatic, Frame -> True]
```

Mathematica can also calculate the pseudoinverse of a symbolic matrix. Because the resulting matrix for a 3×2 input matrix is quite large, we extract common denominators using `extractCommonDenominator`.

```
extractCommonDenominator[m_?MatrixQ] :=
Module[{(* the common denominator *)
        den = PolynomialLCM @@ Denominator[Flatten[Simplify[m]]]},
       (HoldForm @@ {Cancel[m den]})/den /.
        (* use bar for conjugation *) Conjugate[a_] :> OverBar[a]]

PseudoInverse[Table[Subscript[a, i, j], {i, 3}, {j, 2}]] //
                                        extractCommonDenominator
```

Most of the commands relating to linear algebra introduced in this chapter possess options.

```
Options[Det]

Options[Inverse]

Options[Eigensystem]

Options[Eigenvectors]

Options[Eigenvalues]

Options[LinearSolve]
```

The option `Modulus -> ` *integer* is of no interest here; it essentially says that all numbers that appear are to be regarded modulo *integer*. The two other options are `Method` and `Inverse`.

`Method`

is an option for the commands `LinearSolve`, `Inverse`, `RowReduce` (to be treated soon), and `NullSpace`. It defines the internal algorithm to be used in the computation.

Default:

`Automatic`

Admissible:

`DivisionFreeRowReduction` or `CofactorExpansion` or `OneStepRowReduc·tion`

It is not easy to give general guidelines for deciding which method to use for which kind of matrices (sparse or full; symbolic, exact, or numerical; the ratio of the largest/smallest element; etc.). The speed of execution and size of the result (for underdetermined systems of equations, even the result itself) may depend heavily on the method used. Thus, the user should explore the various methods for the matrices at hand. Here is an example for `LinearSolve`.

```
Clear[a, b, c, M, vec];
M = {{a, 1, b, 2}, {c, 0, a, 1}, {a, a, 2, 0}, {0, 2, a, b}};
vec = {1, 1, 1, 1};
```

```
Timing[ByteCount[LinearSolve[M, vec, Method -> #]]]& /@
  {CofactorExpansion, DivisionFreeRowReduction,
   OneStepRowReduction, Automatic}
```

Be aware that not only the timings but also the explicit form of the results depend on the chosen method.

```
SameQ @@ (LinearSolve[M, vec, Method -> #]& /@
  {CofactorExpansion, DivisionFreeRowReduction,
   OneStepRowReduction, Automatic})
```

The second option is `ZeroTest`.

`ZeroTest`

is an option for the commands `Eigensystem`, `Eigenvectors`, `LinearSolve`, `Inverse`, `RowReduce` (to be treated below), and `NullSpace`. It defines the function to be applied to determine whether matrix elements and temporary expressions are zero.

Default:

for `LinearSolve`, `Inverse`: `(# == 0&)` for `Eigensystem`, `Eigenvectors`, `NullSpace`, `RowReduce`: `Automatic` (meaning various heuristic tests)

Admissible:

arbitrary (pure) function or `Automatic`

Here is an obviously singular matrix.

```
nullMatrix = (* 4 hidden zeros *)
{{x(x + 1) - (x^2 + x), Cos[1]^2 - 1/2(1 + Cos[2])},
 {Sin[1] - 2 Sin[1/2] Cos[1/2], Sin[Pi/8] - Sqrt[2 - Sqrt[2]]/2}}
```

We can see that all elements are zero by using `FullSimplify` (we discuss `FullSimplify` in detail in Chapter 3 of the Symbolics volume [303✱]).

```
FullSimplify[nullMatrix]
```

With the default `ZeroTest -> (# == 0&)`, we seem to get an inverse.

```
Inverse[nullMatrix]
```

But this nonsingularity only seems to be the case.

```
N[%]
```

With the setting `ZeroTest -> Automatic`, `Inverse` recognizes that it is dealing with a singular matrix during the computation.

```
Inverse[nullMatrix, ZeroTest -> Automatic]
```

To illustrate the application of mathematical operations on lists, we give one more example, the so-called quantum cellular automata.

Physical Remark: Quantum Cellular Automata

Suppose we are given a list $c_{0\,j}$ ($j = 1, \ldots, n$) of complex numbers (the states of the individual particles (=elements of a discretization of a function describing them) in a physical system at time $t = 0$). For $j < 1$ and $j > n$, we continue the list periodically:

$$c_{0\,n+1} = c_{0\times 1},\ c_{0\,n+2} = c_{0\times 2},\ \ldots,\ c_{0\times 0} = c_{0\,n},\ c_{0\,-1} = c_{0\,n-1},\ \ldots$$

The state of the system c_{ij} at a later time (we consider only discrete time steps here) is given by

$$c_{i+1\,j} = \mathcal{N}\left(c_{i\,j} + i\,\delta\, c_{i\,j-1} + i\,\overline{\delta}\, c_{i\,j+1}\right),\ c_{i\,n+1} = c_{i\,1},\ c_{i\,0} = c_{i\,n}.$$

Here δ is a complex parameter characterizing the system, and $\mathcal{N}$ is a normalization constant defined implicitly so that we have for all i

$$\sum_{j=1}^{n} \left|c_{i\,j}\right|^2 = 1.$$

For details on quantum cellular automata, see [118✱], [119✱], [116✱], [117✱], [5✱], [30✱], [210✱], [147✱], [45✱], [13✱], and [95✱]; and for a general treatment on cellular automata, see, for example, [327✱]. For quantum random walks, see [224✱].

We now want to find the c_{ij} ($i = 1, \ldots, m$) for a given list $c_{0\,j}$. Here is an implementation. Note that we can get by without using temporary auxiliary variables. First, for each list, we add the last element to the front and the first element to the end of the list, as suggested by the above periodicity condition. The resulting list is divided into sublists of length three using `Partition[..., 3, 1]`, and then the elements at the next level are computed using `Dot[{I d, 1, I Conjugate[d]}, #]& /@ ...`. Finally, the function `Function[p, p/Sqrt[p.Conjugate[p]]]` is used to compute $\mathcal{N}$, and all elements are divided by $\mathcal{N}$ = `Sqrt[p.Conjugate[p]]`. This process is repeated `iter` times via `NestList`.

```
QuantumCellularAutomata[start_, δ_, iter_] :=
NestList[Function[p, p/Sqrt[p.Conjugate[p]]][
            ({I δ, 1, I Conjugate[δ]}.#& /@
               Partition[Prepend[Append[#, First[#]], Last[#]], 3, 1])]&,
         Function[p, p/Sqrt[p.Conjugate[p]]][N[start]], iter]
```

Symbolically, this result grows very quickly, while a numerical example is much faster and shorter.

```
{LeafCount[#], ByteCount[#]}&[
              QuantumCellularAutomata[{α, β, γ, δ, ϵ}, 2, 3]]

QuantumCellularAutomata[{0, 0, 1, 0, 0}, 2, 5]
```

This implementation allows us to choose significantly longer initial lists, and to use many more iterations, whereas still only using an acceptable amount of time. We now look at the resulting data sets graphically. To get real numbers, we use `Abs[#^2]&`. We discuss the command `ListDensityPlot` and its options in detail in Chapter 3 of the Graphics volume [301✱].

```
ListDensityPlot[Abs[Transpose @ QuantumCellularAutomata[
    Join[#, {1}, #]&[Table[0, {70}]], 20(1 + I), 1000]]^2,
                    (* color according to the absolute value *)
                    ColorFunction -> (Hue[0.74#]&), Mesh -> False,
                    Frame -> False, AspectRatio -> 1/3] // Timing
```

For some interesting patterns formed by friends of quantum cellular automata, namely heads of quantum Turing machines, see [161✱], and [162✱].

To end this subsection, we give some comments regarding "symbolic" matrix calculations. By symbolic, we mean that the matrix is not explicitly given, so its dimensions are not known. Let us start with solving a matrix equation.

```
Clear[A, b, x];
Solve[A.x == b, x]
```

The result looks a bit strange at first sight, but is reasonable. *Mathematica* does not give `Dot` special treatment, so it just says that we should take the inverse with respect to the second argument. We could bring this into a more common form by making a definition. (We use `Dot[a]` on the right-hand side of the following definition to match also the cases `A.B.x == b`, `A.B.C.x == b`, ...)

```
Unprotect[InverseFunction]

InverseFunction/:
InverseFunction[Dot, n_, n_][a__, b_] := Inverse[Dot[a]].b
```

Now, we have the following behavior. (Do not worry about the warning; it just means that whenever inverse functions are used, some possible solutions may be lost. However, we know this will not be the case in this example, because we are inverting a linear relation. And because the expression `A.Inverse[A]` for a symbol `A` without a value cannot evaluated to an identity matrix of unknown dimension, it will stay unevaluated. As a result, `Solve` cannot verify that the found solution was correct and will discard it. The option setting `VerifySolutions -> False` tells `Solve` to skip the verification step.)

```
Solve[A.x == b, x, VerifySolutions -> False]
```

Now, let us look at a slightly more complicated example.

```
Solve[A.B.x == b, x, VerifySolutions -> False]
```

Again, we had only partial success in our first trial. Probably, we would like to see `Inverse[A.B]` "done". We can easily attach a corresponding rule to `Inverse`.

```
Unprotect[Inverse]

Inverse[matProd_Dot] := Dot @@ (Inverse /@ Reverse[List @@ matProd])
```

Here is our result.

```
Solve[A.B.x == b, x, VerifySolutions -> False]
```

Let us remove the above definitions. They show that not much is built-in for symbolic matrix manipulations, but it is no problem to add the missing definitions to the built-in rules to get the desired behavior.

```
Clear[InverseFunction, Inverse]

Protect[InverseFunction, Inverse]
```

Numerical linear algebra with sparse matrices we will discuss in Chapter 1 of the Numerics volume [302✱].

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

■ 6.5.2 Constructing and Solving Magic Squares

As an application of lists and the linear algebra commands in *Mathematica*, in this subsection, we construct magic squares and solve them. We take a rather naive and straightforward approach; for a more mathematical construction, see the references cited below. A magic square is a square array of positive integers so that the sum of the elements in its columns is equal to the sum of the elements in its rows and to the sum of its elements along its `mainDiagonal` and `subDiagonal`. (Sometimes, it is also required that each number appear only once in the magic square; we do not demand this here.) Note that a magic square of nth order contains n^2 elements, but that the number of equations that determine its elements is only $2\,n+2$; the system of equations is underdetermined for $n > 2$. Using `LinearSolve`, we get a good solution in the sense that only relatively small numbers occur.

We begin with the construction of magic squares. In order to apply `LinearSolve`, we need to find the coefficient matrix of the corresponding system of equations. We consider every element of the magic square to be an unknown, and number the unknowns row by row. Thus, for $n = 4$, we have:

$$\begin{matrix} x_1 & x_2 & x_3 & x_4 \\ x_5 & x_6 & x_7 & x_8 \\ x_9 & x_{10} & x_{11} & x_{12} \\ x_{13} & x_{14} & x_{15} & x_{16} \end{matrix}$$

The coefficient matrix for the $2\,n+2$ equations for a magic square of nth-order can be constructed as follows.

```
equationsMagicSquare[n_Integer] :=
 Module[{rows, columns, mainDiagonal, subDiagonal},
   (* n left-hand sides of the equation for the n rows *)
   rows = Flatten /@ Table[If[i == j, Table[1, {n}],
                                      Table[0, {n}]], {j, n}, {i, n}];
   (* n left-hand sides of the equation for the n columns *)
   columns = Flatten /@ Partition[Transpose[rows], n];
   (* equation for the main diagonal *)
   mainDiagonal = Flatten[Table[If[i == j, 1, 0], {i, n}, {j, n}]];
   (* equation for the subDiagonal *)
   subDiagonal = Flatten[Table[If[i == j, 1, 0], {i, n, 1, -1}, {j, n}]];
   (* combine the 2n + 2 equations *)
   Join[rows, columns, {mainDiagonal, subDiagonal}]]
```

We now look at the resulting rectangular coefficient matrices for $n = 3$ and $n = 4$.

```
TableForm[equationsMagicSquare[3], TableSpacing -> {1, 1}]
```

```
TableForm[equationsMagicSquare[4], TableSpacing -> {1, 1}]
```

Next, we look for a solution of this system of equations. We choose one "free parameter", the value of the sums of the `rows`, `columns`, `mainDiagonal`, and `subDiagonal`. We use this value only temporarily; the resulting magic square will have a different sum for its `rows`, `columns`, `mainDiagonal`, and `subDiagonal`. The reason is that `LinearSolve` also produces negative fractions as solutions. Because magic squares usually consist only of positive integers, we multiply all elements with the least common multiple of the denominators, and add two to the absolute value of the smallest negative element to eliminate negative elements. We could have used any other transformation that ensures positivity of all elements. These operations do not affect the equality of the sums of the rows, columns, maindiagonal, and subdiagonal, but only the numerical value of this sum. To be able to study the intermediate results later, the local variables of `Module` are enclosed in comment brackets. The function `LCM` calculates the least common multiple of a set of number; we will discuss it in more detail in Chapter 2 of the Numerics volume [302✱].

```
magicSquare[n_Integer, (* size *)
            rightHandSide_Integer | rightHandSide_Rational
            (* "the parameter sum" *)] :=
Module[{(* the local variables are in a comment to
        see their values outside of the Module *)
          (* sol1, sol2, sol3, magical, summe *)},
(* find a special solution of the underdetermined system of equations *)
sol1 = LinearSolve[equationsMagicSquare[n],
                   Table[rightHandSide, {2n + 2}]];
(* multiply this solution with the least common multiple
  [formed with LCM] of the denominators [extracted with Denominator] *)
sol2 = (LCM @@ Denominator /@ sol1) sol1;
(* add the smallest negative element + 2, or 2, respectively *)
sol3 = sol2 + If[Min[sol2] < 0, -Min[sol2] + 2, 2];
(* partition the sequence of elements obtained above
  into rows of length n *)
magical = Partition[sol3, n];
(* compute the sum of the rows, columns, mainDiagonal and subDiagonal *)
sum = Plus @@ magical[[1]];
(* Output the magic square itself, and the
  sum of the rows, columns, main-, and subDiagonals *)
 {magical, sum}]
```

Here are a few examples. We use `TableForm` instead of `MatrixForm` because `magicSquare` is not a rectangular matrix.

```
magicSquare[3, 5] // TableForm

magicSquare[3, 22] // TableForm

magicSquare[3, 3/7] // TableForm

magicSquare[4, 23/17] // TableForm

magicSquare[5, 0] // TableForm
```

We now look at a special example to examine the computational steps.

```
magicSquare[3, 5]
```

`sol1` is a solution of the system of linear equations.

```
sol1
```

`sol2` arises from `sol1` by multiplication with the least common multiple (computed with `LCM`) of its denominators. `sol2 = (LCM @@ Denominator /@ sol1) sol1`.

```
sol2
```

We get `sol3` from `sol2` by adding either 2 or 2 + *absoluteValueOfSmallestElement*: `sol3 = sol2 + If[Min[sol2] < 0, -Min[sol2] + 2, 2]`.

```
sol3
```

Then, `magical` is created by partitioning the sequence of elements in `sol3` into rows of length *n*. `magical = Partition[sol3, `*n*`]`.

```
magical
```

`sum` is found by computing the sum of the rows, columns, maindiagonal, and subdiagonals: `sum = Plus @@ magical[[1]]`.

```
sum
```

To make this example into a puzzle, we need to code our magic square. We identify for instance `0` with `A`, `1` with `B`, `2` with `F`, `3` with `G`, `4` with `H`, `5` with `J`, `6` with `K`, `7` with `L`, `8` with `M`, and `9` with `P`.

```
codedMagicSquare[n_Integer,
                  rightHandSide_Integer | rightHandSide_Rational] :=
Module[{(* working variable *) aux},
 (* computation of the magic square and removal of its inner brackets
  to simplify later computations;
  here we could have used a Map[..., ..., {-1}] construction *)
 aux = Flatten[magicSquare[n, rightHandSide], 2];
 (* transform the numbers to lists of strings of the individual digits *)
 aux = Characters[ToString[#]]& /@ aux;
 (* replace the digits by letters *)
 aux = aux //. {"0" -> "A", "1" -> "B", "2" -> "F", "3" -> "G", "4" -> "H",
                "5" -> "J", "6" -> "K", "7" -> "L", "8" -> "M", "9" -> "P"}
 (* combine the individual letters *)
 aux = (StringJoin @@ #)& /@ aux;
 (* build the original form {{magic square}, sum} *)
 {Partition[aux, n], Last[aux]}]
```

Finally, we have a true magic square.

```
codedMagicSquare[3, 5] // TableForm
```

We now look at the converse: Given a magic square (or a related puzzle) and its sum in the form of coded letters, find the numbers associated with the letters. To this end, we first define a function `toNumber` that converts a string into a sum of the products of the letters with 10^n.

```
toNumber[s_String] :=
Module[{ch}, ToExpression[chars = Characters[s]].
             Table[10^i, {i, Length[chars] - 1, 0, -1}]];
```

Here is an example with a rather long sequence of letters.

```
toNumber["AABMPPQRSTXYZZZZ"]
```

In a certain sense, the solution of a magic square can be more difficult than its construction. Thus, we first program a preliminary step: `preSolveMagicSquare`. This routine solves the equations for a given magic square; however, in general, the solutions are neither positive integers nor free of arbitrary parameters.

```
preSolveMagicSquare[magic_List] :=
Module[{aux, vars, magicS, magigSN, dim, eqns},
  (* auxiliary variables *)
  aux = Union[Flatten[Characters /@ Flatten[magic]]];
  (* create the desired letters as symbols *)
  vars = ToExpression /@ aux;
  (* extract the magic square and the sum of the
   rows, columns, main-, and subDiagonals *)
  magicS = magic[[1]]; sum = magic[[2]]; dim = Length[magicS];
  (* convert the sequence of letters to coefficients and powers of 10 *)
  magicSN = Map[toNumber, magicS, {2}];
  (* combine the equations *)
  eqns = Join[(Plus @@ #)& /@ magicSN,
              (Plus @@ #)& /@ Transpose[magicSN],
              {Sum[magicSN[[i, i]], {i, dim}],
               Sum[magicSN[[dim - i + 1, i]], {i, dim}]}];
  (* convert the sequence of letters in the sums of the rows, columns,
   main-, and subDiagonals into coefficients and powers of 10 *)
 sum = toNumber[sum];
  (* connect the left-hand and right-hand sides
   of the equations to each other *)
 eqns = Equal[#, sum]& /@ eqns;
 (* solve the system of equations *)
 Solve[eqns, vars]]
```

We now attempt to solve the magic square constructed above.

```
test = codedMagicSquare[3, 5]

preSolution = preSolveMagicSquare[test]
```

Here is the usual problem with magic squares. The system of equations arising from the sums of the rows, columns, main diagonals, and subdiagonals does not suffice to uniquely determine the digits associated with the letters (see the above discussion of the number of equations in a magic square). That is why we used `Solve` in `preSolveMagic⋱ Square` rather than `LinearSolve` to find a solution to the system of equations. We obtain the undetermined variables by sorting out all objects with the head `Symbol` on the right-hand side of the replacement rules produced by `Solve`.

```
variables = Union[Cases[Level[#[[2]]& /@ preSolution[[1]], {-1}], _Symbol]]
```

We still have to sort out the integer solutions for the desired letters. To do this, we first convert the list of the "parameter letters" obtained above into a list of iterators. Using `Sequence`, we can apply this "conjoining" of lists in a `Do` loop, for example.

```
iterators = {#, 0, 9}& /@ variables

iterators = Sequence @@ iterators
```

We now insert these iterators into the solution found above and check for integers.

```
Do[If[And @@ (IntegerQ /@ (#[[2]]& /@ preSolution[[1]])),
      Print[Sequence @@ variables]], Evaluate[iterators]]
```

We now package this code. The following function `solveMagicSquare` gives all possible identifications *letters→int⋅ egers*. It allows one digit to be mapped to different letters.

```
SolveMagicSquare[magic_List] :=
Module[{preSolution, variables, varString, iterators},
  (* a first solution, maybe containing free parameters *)
  preSolution = preSolveMagicSquare[magic];
  (* the variables still present in preSolution *)
  variables = Union[Cases[Level[#[[2]]& /@
                  preSolution[[1]], {-1}], _Symbol]];
  (* stringified variables *)
  varString = ToString[variables];
Which[Length[variables] == 0,
       CellPrint[Cell[TextData[{"∘ All Variables are determined."}],
                      "PrintText"]],
      Length[variables] >= 1,
       CellPrint[Cell[TextData[{
             "∘ The following variables remain undetermined: ",
                    StyleBox[varString, "MR"]}], "PrintText"]]];
CellPrint[Cell[TextData[{"∘ The possible solutions are:"}], "PrintText"]];
  (* calculate all possible solutions *)
 (* the iterators over the variables *)
  iterators = Sequence @@ ({#, 0, 9}& /@ variables);
  solnList = Flatten[Append[(ToString[#[[1]]] -> #[[2]])& /@
                                            preSolution[[1]],
                 (ToString[#] -> #)& /@ variables]];
  solution = {};
  (* collect all possible solutions *)
  Do[(* check solution *)
     If[And @@ ((IntegerQ[#] && 0 <= # <= 9)& /@
                 (#[[2]]& /@ preSolution[[1]])),
              AppendTo[solution, solnList]],
     Evaluate[iterators]];
  (* return the solutions *)
   solution]
```

Once again, we solve the magic square constructed above.

```
SolveMagicSquare[codedMagicSquare[3, 5]]
```

The second solution is our above encoding.

Finally, we give one last example to test `solveMagicSquare`.

```
magicSquare[4, 34/78] // TableForm

codedMagicSquare[4, 34/78] // TableForm

SolveMagicSquare[%]
```

We compare it again with our coding.

```
{"0" -> "A", "1" -> "B", "2" -> "F", "3" -> "G", "4" -> "H",
 "5" -> "J", "6" -> "K", "7" -> "L", "8" -> "M", "9" -> "P"};
```

Two reasons explain why the same letter always appears in the lower right corner.

First, the equations for determining the numbers in a magic square are not linearly independent, although there are only a small number of them in comparison with the number of unknowns, which can be seen using `RowReduce`.

`RowReduce[`*rectangularMatrix*`]`

constructs a simplified form of *rectangularMatrix* by taking linear combinations of the rows and columns.

For a 3×3 magic square, all equations are still linearly independent.

```
equationsMagicSquare[3] // MatrixForm
```

However, for a 4×4-magic square, they are no longer linearly independent.

```
RowReduce[equationsMagicSquare[3]] // MatrixForm
```

The second reason relates to the first. The same letter appears in the lower right corner of the magic square because of the sorting behavior of `LinearSolve`. If we had used `Solve` in the generation of magic squares in an analogous way, we would have been able to build a much wider variety of magic squares.

With a little effort, it is possible to use `Solve` to find the principal structure of a magic square of *n*th-order. Here is an example with $n = 3$ and sum $3\,A$.

```
Partition[
   (({a1, a2, a3, a4, a5, a6, a7, a8, a9} /.
      Solve[ ({a1, a2, a3, a4, a5, a6, a7, a8, a9}.# == 3/2a)& /@
                equationsMagicSquare[3],
                 {a1, a2, a3, a4, a5, a6, a7, a8, a9}]) /.
                  (* write in nicer form *)
                  {{a -> 2A, a7 -> B, a8 -> A + B - C, a9 -> A + C}} //
       Simplify)[[1, 1]], 3] // TableForm[#, TableAlignments -> Center]&
```

It is also possible to find the replacement rules needed here, but this is somewhat complicated; see the references cited below.

Now, we look at a “real” magic square-like puzzle (adapted from [334✱]): The letters in the square

```
UH  EE  HU  LR  ÖG
GU  ÖR  AG  EH  HE
SG  HH  GE  RU  AR
RE  UU  SR   G  GH
 R  LG  RH  UE  EU
```

are to be replaced by integers so that the same letters have the same integers, and different letters are to be replaced by different integers. In addition, the sums of the five numbers in every column should be the same as the sum of the five numbers in each of the two diagonals, namely, the value represented by ÖEE. Moreover, if the numbers are put in increasing order, each successive pair should differ by the same constant. (If corresponding letters are put in the order corresponding to the increasing numbers, they give the name of the author’s home town.)

```
Short[#, 6]& @
(sol = SolveMagicSquare[
        {{{"uh", "ee", "hu", "lr", "ög"},
          {"gu", "ör", "ag", "eh", "he"},
          {"sg", "hh", "ge", "ru", "ar"},
          {"re", "uu", "sr",  "g", "gh"},
          { "r", "lg", "rh", "ue", "eu"}}, "öee"}])
```

Because of the many possible interpretations, we do not write them all out; we do collect them in `sol` for later use. The 37 solutions arise from various interpretations of the letters as numbers.

```
Length[sol]
```

The solution we want is determined by the condition that if the numbers are put in increasing order, each successive pair should differ by the same constant. Here, this condition is coded.

```
Select[sol, (Length[Union[Apply[(#2 - #1)&, #]& /@
                Partition[Sort[#[[2]]& /@ #], 2, 1]]] == 1)&]
```

The last step automates the computation of the solution word and capitalizes the first letter.

```
makeWord[li_] :=
StringJoin[(* capitalize first letter *)
           ToUpperCase[StringTake[#, 1]], StringDrop[#, 1]]&[
  StringJoin[Function[x, x[[1]], (* sort *)
           {Listable}][Sort[li, #1[[2]] < #2[[2]]&]]]]
```

This substitution gives the correct solution.

```
{makeWord[%%[[1]]], makeWord[%%[[2]]], makeWord[%%[[3]]]}
```

So the answer is `Hörselgau` (located in Thuringia at the foot of the Hörselberg mountain chain (the reader might know them from Richard Wagner's *Tannhäuser* opera) and the foot of the Inselsberg (one of the mountains Gauss used to measure the sum of angles in a geographical triangle [267✱]).

For more on magic squares, see [15✱], [173✱], [55✱], and [6✱]. For some deeper number theory studies of magic squares, see [28✱], [270✱], [171✱], [330✱], [312✱], [144✱], [274✱], [40✱], [259✱], [286✱], [36✱], [275✱], [3✱], [44✱], [78✱], [308✱], [131✱], [34✱], and [132✱]. Magic hexagons are treated in [133✱], magical parquets in [23✱], and magic cubes in [4✱] and [254✱]. For inertia tensors of "massified" magic squares, see [258✱].

In a similar way, we could implement solutions to problems like the following [325✱]: Replace each of the letters in the following sum by digits, such that the addition becomes correct: `GAUSS+RIESE=EUKLID`.

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

■ 6.5.3 Powers and Exponents of Matrices

The operations discussed in the previous subsection are all linear. It is also possible to compute powers of matrices.

`MatrixPower[`*matrix*`,` *exponent*`]`
gives the *exponent*th power of the square matrix *matrix*.

Roughly speaking, a function f of a matrix is defined by the Taylor (Laurent) series of the function f, with powers replaced by iterated matrix products. (For mathematical details on the definition of functions of square matrices, see [193✱], [17✱], [257✱], [188✱], [182✱], and [256✱].) Here is a rather large power. Here is the 100th power of an integer-values 2×2 matrix.

```
MatrixPower[{{1, 2}, {3, 4}}, 100]
```

The matrix power $\mathbf{A}^{-1}$ is just the inverse of the matrix $\mathbf{A}$. The following input demonstrates this for a generic 2×2 matrix.

```
MatrixPower[{{a11, a12}, {a21, a22}}, -1] ==
    Inverse[{{a11, a12}, {a21, a22}}]
```

Using the Taylor series $\left(matrix^n / n!\right)$, it is also possible to define e^{matrix} [219✱], [127✱], [248✱], [67✱].

`MatrixExp[`*matrix*`]`
gives the value e^{matrix} of the exponential function applied to the square matrix *matrix*.

Here is e^{matrix} of the above matrix.

```
MatrixExp[{{1., 2.}, {3., 4.}}]
```

Using `FixedPointList`, we can examine how this result comes about. We have *matrix*`^0` = **1**, where **1** is the identity matrix of the same dimension as *matrix*.

```
n = 0;
FixedPointList[
(n = n + 1; # + MatrixPower[{{1., 2.}, {3., 4.}}, n]/n!)&,
                {{1., 0.}, {0., 1.}}] // Short[#, 12]&
```

Thus, a total of 35 iterations are needed to obtain machine accuracy.

```
Length[%]
```

Here, `MatrixExp` is used for a numerical check of the identity $\det\left(e^{\mathbf{A}}\right) = e^{\operatorname{Tr}\mathbf{A}}$. For **A**, we use a Hilbert matrix with elements $(i + j + 1)^{-1}$, and the matrix dimension ranges from 1 to 5.

```
Table[Det[MatrixExp[#]] -
      Exp[Plus @@ MapIndexed[Take, #]]&[
                   Array[N[1/(#1 + #2 + i)]&, {8, 8}]], {i, 1, 5}]
```

```
Table[Det[MatrixExp[#]] - Exp[Tr[#]]&[
                    Array[N[1/(#1 + #2 + i)]&, {8, 8}]], {i, 1, 5}]
```

Using an input matrix with high-precision numbers as elements shows that the identity holds within the precision of the calculation.

```
Table[Det[MatrixExp[#]] - Exp[Tr[#]]&[
                    Array[N[1/(#1 + #2 + i), 30]&, {8, 8}]], {i, 1, 5}]
```

Similar to the exponential function of a scalar argument, we can have exp(**A**) = exp(**B**) for two matrices **A**, and **B** with $\mathbf{A} \neq \mathbf{B}$. Here is an example [268✱].

```
MatrixExp[Pi {{0, -1}, {1, 0}}] === MatrixExp[Pi {{1, 1}, {-2, -1}}]
```

Next, we define an $n \times n$ integer-valued matrix with the integers 1, 2, …, $n - 1$ below the diagonal and 0 else [2✱].

```
𝒜𝒫[n_] := Table[If[j == i - 1, i, 0], {i, 0, n - 1}, {j, 0, n - 1}];
```

The exponential function of `𝒜𝒫[n]` can be calculated through its defining series. The series terminates after the *n*th term. The function `fillInStageExp𝒜𝒫` marks through which term an element gets filled.

```
fillInStageExp𝒜𝒫[d_] :=
Plus @@ (MapIndexed[(* mark elements *) C[#2[[1]], #1/(#2[[1]] - 1)!]&,
        Drop[FixedPointList[𝒜𝒫[d].#&, IdentityMatrix[d]], -1], {3}] /.
        C[_, 0_] :> 0) /. (* keep fill-in time *) C[s_, _] :> s
```

Here is the matrix `𝒜𝒫[6]`, its exponential function, and the fill in-time of the elements.

```
MatrixForm /@ {𝒜𝒫[6], MatrixExp[𝒜𝒫[6]], fillInStageExp𝒜𝒫[6]}
```

Using `MatrixExp`, we can implement, for instance, a matrix version of `Cos`.

```
MatrixCos[m_] := (MatrixExp[I m] + MatrixExp[-I m])/2
```

It is well known that iterating `Cos` yields a fixpoint, the solution of `Cos[x] == x`.

```
FixedPoint[Cos, 1.]
```

```
Cos[%] - %
```

Here, the same is done for our `MatrixCos` and a "random" starting matrix.

```
startMat[n_] := Table[1/(i + j^2 + 3.), {i, n}, {j, n}];
```

```
fpl[n_] := FixedPointList[MatrixCos, startMat[n],
                          SameTest -> (Max[Abs[#1 - #2]] < 10^-10&)];
```

For a 1D matrix, we recover the result from above.

```
fpl[1] // Short[#, 4]&
```

For a 2D matrix, we obtain a diagonal matrix with the entries above.

```
(fl = fpl[2]) // {Length[#], Last[#]}&
```

Again, we found a solution of `Cos[x] == x`.

```
MatrixCos[Last[fl]] - Last[fl] // Chop
```

Here, the convergence is visualized. We display the logarithmic difference as a function of the number of iterations.

```
ListPlot[Log[10, (Max[DeleteCases[Abs[#],
                              (* no zeros *) _?(# == 0.&), {-1}]])&  /@
                  Apply[Subtract, Partition[fl, 2, 1], {1}]],
         PlotRange -> All, Frame -> True]
```

Using high-precision numbers instead of machine numbers shows that the spurious imaginary parts in the last results are really zero. To avoid loosing precision at each step, we use the function `SetPrecision`.

```
startMat[n_] := Table[N[1/(i + j^2 + 3), 30], {i, n}, {j, n}];

fpl[n_]  := FixedPointList[SetPrecision[MatrixCos[#], 20]&,
                           startMat[n], 1000, SameTest -> Equal]

(fl3 = fpl[3]) // {Length[#], Last[#]}&
```

The nondiagonal elements approach zero, but are always nonzero. So the `Equal` same test yields always `False` and the iterations stop after *maxIter*=1000 steps. The following graphic show how the diagonal (red points) approach a fixed point and how the nondiagonal matrix elements (blue points) shrink. (Because we work with 20-digit matrices, the diagonal differences become effectively zero after about 100 iterations.)

```
Module[{pairDifferences, diagonalDifferences, nonDiagonalDifferences},
 (* matrix differences *)
 pairDifferences = (Subtract @@@ Take[Partition[fl3, 2, 1], All]);
 (* maximal diagonal and nondiagonal matrix element differences *)
 diagonalDifferences = Max[Abs[IdentityMatrix[3] #]]& /@ pairDifferences;
 nonDiagonalDifferences = Max[Abs[(1 - IdentityMatrix[3]) #]]& /@
                                                         pairDifferences;
 (* show diagonal differences in red and nondiagonal differences in blue *)
 Show[Graphics[{PointSize[0.003],
  Function[{col, data}, {col, Point /@ DeleteCases[
   MapIndexed[{#2[[1]], Log[10, #1]}&, data], {_, Indeterminate}]}] @@@
   {{RGBColor[0, 0, 1], nonDiagonalDifferences},
    {RGBColor[1, 0, 0], diagonalDifferences}}}],
    Frame -> True, PlotRange -> All]]
```

`MatrixExp` can also deal with matrices containing symbolic inputs. Such matrix exponentials arise frequently when dealing with Lie groups. Here is a typical example [318✱].

```
m = MatrixExp[1/2 {{0, 0, ωz, ωx - I ωy},
                   {0, 0, ωx + I ωy, -ωz},
                   {ωz, ωx - I ωy, 0, 0},
                   {ωx + I ωy, -ωz, 0, 0}}];
```

Because no automatic simplification is carried out by `MatrixExp`, such results are typically quite large.

```
LeafCount[m]
```

Simplifying the result yields a compact answer.

```
FixedPoint[Simplify[# //.
                   (* let ω be the norm of the vector {ωx, ωy, ωz} *)
                   f_. ωx^2 + f_. ωy^2 + f_. ωz^2 :> f ω^2 //.
                   {Sqrt[-ω^2] :> I ω, 1/Sqrt[-ω^2] :> 1/(I ω),
                    Sqrt[ω^2] :> ω, 1/Sqrt[ω^2] :> 1/ω}]&, m]
```

`MatrixPower` also works with noninteger exponents.

Here is the square root of a matrix.

```
MatrixPower[{{1., 2.}, {3., 4.}}, 1/2]
```

If we square it, we get the original matrix again.

```
%.% // Chop
```

It is also possible to find roots of a symbolic matrix.

```
MatrixPower[{{a, b}, {c, d}}, 1/2]
```

Here is the third root of a numerical matrix.

```
MatrixPower[{{1., 2.}, {3., 4.}}, 1/3]
```

Multiplying three copies of this matrix together gives back the original matrix (within roundoff error).

```
%.%.%
```

Here the same operation is carried out using high-precision numbers.

```
#.#.#&[MatrixPower[N[{{1, 2}, {3, 4}}, 100], 1/3]] - {{1, 2}, {3, 4}}
```

Using the spectral decomposition of the matrix generated by `Eigensystem`, we can calculate the third root "by hand". If **C** is the transposed matrix of the eigenvectors (fulfilling $\mathbf{C}.\mathbf{C}^{-1} = \mathbf{C}^{-1}.\mathbf{C} = \mathbf{1}$), which diagonalizes the matrix **A** (this means $\mathbf{C}^{-1}.\mathbf{A}.\mathbf{C}$ is a diagonal matrix; and taking the third root of it just reduces to taking the third root of the elements from the diagonal), we have

$$\begin{aligned}\mathbf{A} &= \mathbf{C}.\mathbf{C}^{-1}.\mathbf{A}.\mathbf{C}.\mathbf{C}^{-1}\\ &= \mathbf{C}.\left(\mathbf{C}^{-1}.\mathbf{A}.\mathbf{C}\right)^{1/3}.\left(\mathbf{C}^{-1}.\mathbf{A}.\mathbf{C}\right)^{1/3}.\left(\mathbf{C}^{-1}.\mathbf{A}.\mathbf{C}\right)^{1/3}.\mathbf{C}^{-1}\\ &= \mathbf{C}.\left(\mathbf{C}^{-1}.\mathbf{A}.\mathbf{C}\right)^{1/3}.\mathbf{C}^{-1}.\mathbf{C}.\left(\mathbf{C}^{-1}.\mathbf{A}.\mathbf{C}\right)^{1/3}\,\mathbf{C}^{-1}.\mathbf{C}.\left(\mathbf{C}^{-1}.\mathbf{A}.\mathbf{C}\right)^{1/3}.\mathbf{C}^{-1}\end{aligned}$$

so that $\mathbf{A}^{1/3} = \mathbf{C}.\left(\mathbf{C}^{-1}.\mathbf{A}.\mathbf{C}\right)^{1/3}\,\mathbf{C}^{-1}$ Because $\mathbf{C}^{-1}.\mathbf{A}.\mathbf{C}$ is a diagonal matrix, this quantity is easy to calculate to any power. Here, this identity is exemplified.

```
myPower[mat_, n_] :=
Module[{evals, evecs, C, matDiagonal},
       (* the eigensystem *)
       {evals, evecs} = Eigensystem[mat];
       (* transform matrix to diagonal form *)
       C = Transpose[evecs];
       matDiagonal = Inverse[C].mat.C;
       (* take ordinary power of matDiagonal and transform back *)
       C.Power[matDiagonal, n].Inverse[C]]

myPower[{{1., 2.}, {3., 4.}}, 1/3]

%.%.%
```

Using high-precision numbers shows agreement in all significant digits.

```
myPower[N[{{1, 2}, {3, 4}}, 40], 1/3]
```

```
%.%.%
```

Now let us use the matrix functions discussed for an application.

Mathematical Remark: Abstract Evolution Equations

The solution to the second-order ordinary differential equation $u''(t) = \mathcal{L}\, u(t)$, where $\mathcal{L}$ is a t-independent expression, is given by

$$u(t) = \cosh\big(t \sqrt{\mathcal{L}}\,\big)\, u_0 + \frac{\sinh\big(t \sqrt{\mathcal{L}}\,\big)}{\sqrt{\mathcal{L}}}\, v_0.$$

Here $u_0 = u(0)$ and $v_0 = u'(0)$. If we consider the vector equation $\mathbf{u}''(t) = \mathcal{L}.\,\mathbf{u}(t)$ where $\mathcal{L}$ is now a t- and $\mathbf{u}(t)$-independent operator, the solution can be written in the form [189✱], [80✱]

$$\mathbf{u}(t) = \mathcal{U}.\mathbf{u}_0 + \mathcal{V}.\mathbf{v}_0 = \cosh\big(t \sqrt{\mathcal{L}}\,\big).\mathbf{u}_0 + \sinh\big(t \sqrt{\mathcal{L}}\,\big).\big(\sqrt{\mathcal{L}}\,\big)^{-1}.\,\mathbf{v}_0$$

where now $\mathbf{u}_0 = \mathbf{u}(0)$ and $\mathbf{v}_0 = \mathbf{u}'(0)$. The functions $\cosh\big(t\,\mathcal{L}^{1/2}\big)$, $\sinh\big(t\,\mathcal{L}^{1/2}\big)$, and $\big(\mathcal{L}^{1/2}\big)^{-1}$ of the operator $\mathcal{L}$ are to be interpreted appropriately, for instance, through their power series. (Because $\cosh\big(t\,\mathcal{L}^{1/2}\big)$ and $\big(\mathcal{L}^{1/2}\big)^{-1}$ in the second term of the solution are both functions of $\mathcal{L}$, they commute and their order does not matter.

In the following, we will consider the case where $\mathbf{u}(t)$ is a vector with elements $u_n(t)$ and $\mathcal{L}$ is a matrix. We start by implementing the three matrix functions `MatrixCosh`, `MatrixSinh`, and `MatrixSqrt` through the two built-in functions `MatrixExp` and `MatrixPower`. Operator application and operator composition now become matrix multiplication.

```
MatrixCosh[m_] := (MatrixExp[m] + MatrixExp[-m])/2
MatrixSinh[m_] := (MatrixExp[m] - MatrixExp[-m])/2
MatrixSqrt[m_] := MatrixPower[m, 1/2]
```

As a specific example, we consider Newton's equation of motion for three unit mass particles coupled to each other with springs of unit stiffness [62✱]. The 0th and 4th particles are fixed.

```
eqs = {u[1]''[t] ==         - 2 u[1][t] + u[2][t],
       u[2]''[t] == u[1][t] - 2 u[2][t] + u[3][t],
       u[3]''[t] == u[2][t] - 2 u[3][t]};
```

It is straightforward to extract the matrix $\mathcal{L}$ corresponding to the system `eqs` and to calculate the matrix $\mathcal{U}$. Its elements are relatively complicated functions of t.

```
ℒ = {{-2, 1, 0}, {1, -2, 1}, {0, 1, -2}};
U  = MatrixCosh[t MatrixSqrt[ℒ]] // (* some simplification *)
             Normal // ToRadicals // (Together //@ #)& // Simplify //
             ExpToTrig // Simplify
```

Here is a quick check that the so-found solution fulfills the original differential equations and the initial conditions.

```
sols = Rule @@@ Transpose[{{u[1], u[2], u[3]},
            Function[t, #]& /@ (U.{u[1][0], u[2][0], u[3][0]})}];
```

```
{u[1][0], u[2][0], u[3][0]} /. sols /. t -> 0 // FullSimplify
```

```
(* for a purely symbolic verification:
  FullSimplify[RootReduce //@ (eqs /. sols)] *)
eqs /. sols // N[#, 22]& // Simplify
```

Now let us take a much larger example, namely $o = 51$ nonfixed particles (the 0th and the 52th particle are held fixed). This time we construct a numerical solution only.

```
o = 51;
(* 0th and oth particle are fixed *)
ℒ = Table[Which[n == m, -2, Abs[n - m] == 1, 1, True, 0],
          {n, o}, {m, o}];

s = MatrixSqrt[N[ℒ]];
𝒰[t_] := MatrixCosh[t s];
```

For the initial condition $u_k(t = 0) = \delta_{(o-1)/2,k}$, $u'_k(t) = 0$ (this means at the start only the centermost particle is elongated) we calculate the solutions for 100 times t and display the solution.

```
u0 = Table[If[n == (o - 1)/2, 1., 0.], {n, o}];
ListPlot3D[Append[Prepend[#, 0], 0]& /@
           Table[𝒰[t].u0, {t, 0, 12, 12/100}],
           Mesh -> False, PlotRange -> All] // Timing
```

The last calculation has the drawback that for each t we had to calculate a matrix exponential. While this is relatively quick done, carrying out such an exponentiation hundreds of times can take a while. If we consider together with the time evaluation of $u_n(t)$, the time evolution of $u'_n(t)$ given by [264★]

$$\mathbf{u}'(t) = \sinh\left(t\sqrt{\mathcal{L}}\right).\sqrt{\mathcal{L}}.\mathbf{u}_0 + \cosh\left(t\sqrt{\mathcal{L}}\right).\mathbf{v}_0$$

then we get a time-independent map $\{\mathbf{u}(t+\delta t), \mathbf{u}'(t+\delta t)\} = \mathcal{H}(\delta t).\{\mathbf{u}(t), \mathbf{u}'(t)\}$. This means that the four block matrices forming $\mathcal{H}$ are only dependent on δt and not explicitly on t. So they have only once to be calculated. As a result, the above calculation can be carried out much more efficiently. This time we solve for $0 \le t \le 40$. One nicely sees how the initial distribution reaches the fixed 0th and 52th particles and interferences of the reflected and original oscillations occur. We display the resulting solution as a 3D plot and as a density plot.

```
δt = 0.05;
U  = MatrixCosh[δt s];
V  = MatrixSinh[δt s].Inverse[s];

Up = MatrixSinh[δt s].s;
Vp = MatrixCosh[δt s];

(* initial velocities *)
v0 = Table[0, {n, o}];

{u, v} = {u0, v0};
(data = Append[Prepend[First[#], 0], 0]& /@
          Table[{u, v} = {U.u + V.v, Up.u + Vp.v}, {800}];) // Timing

Show[GraphicsArray[
Block[{$DisplayFunction = Identity},
 {ListPlot3D[data, Mesh -> False, PlotRange -> All],
  ListDensityPlot[data, Mesh -> False, PlotRange -> Automatic,
                  MeshRange -> {{0, o + 1}, {0, 800 δt}},
                  ColorFunction -> (Hue[0.8 #]&)]}]]]
```

For the Green's function of the finite linear chain, see [26★], [60★]. (For problems with time-dependent coefficients, the matrix exponent has to be replaced with a time ordered matrix exponent [170★]; for systems of coupled chains, see [81★].)

We conclude this subsection with an illustration of the theorem of Cayley–Hamilton. (Because of time limitations, we program only the case in which all elements are numbers.)

Mathematical Remark: Theorem of Cayley-Hamilton

Let **A** be a square matrix of dimension d. The associated characteristic polynomial (in λ) is $|\mathbf{A} - \lambda\, \mathbf{1}| = 0$, where **1** denotes the corresponding d-dimensional identity matrix. Substituting for λ in this equation **A** itself, the resulting polynomial equation in the matrix **A** is satisfied.

We restrict the arguments to numeric ones to avoid very time- and memory-consuming calculations.

```
CayleyHamiltonTrueQ[mat_List?
            (* Test whether mat is a square matrix
      containing only numbers *)
            ((MatrixQ[#, NumberQ] && Length[Dimensions[#]] == 2 &&
              Dimensions[#][[1]] == Dimensions[#][[2]]) &)] :=
Module[{dim, characteristicPolynomial, characteristicPolynomialList,
        characteristicPolynomialMatrix, λ},
       (* determine the dimension of the matrix *)
       dim = Length[mat];
       (* compute the characteristic polynomial *)
       characteristicPolynomial = Det[mat - λ IdentityMatrix[dim]];
       (* for ease of manipulation, change the head from Plus to List *)
       characteristicPolynomialList = List @@ characteristicPolynomial;
       (* replace the powers of λ by powers of the matrix *)
       characteristicPolynomialMatrix =
            Replace[#, {a_. λ^n_  :>  a MatrixPower[mat, n],
                        a_. λ     ->  a mat,
                        a_        ->  a IdentityMatrix[dim]}]& /@
                                        characteristicPolynomialList;
       (* simplify and check whether the zero matrix is obtained *)
       If[Simplify[Plus @@ characteristicPolynomialMatrix] ==
                                            Table[0, {dim}, {dim}],
           True, False]]
```

Here is a test with a 2×2 matrix.

```
CayleyHamiltonTrueQ[{{5, 3}, {6, 4}}]
```

For nonsquare matrices, no suitable rule is implemented.

```
CayleyHamiltonTrueQ[{{5, 3}, {6, 4}, {8, 9}}]
```

The same holds when symbolic elements appear.

```
CayleyHamiltonTrueQ[{{5, 3}, {6, symbolic}}]
```

Next, we consider a larger matrix.

```
CayleyHamiltonTrueQ[Table[i + j, {i, 6}, {j, 6}]]
```

In the implementation of `CayleyHamiltonTrueQ` given above, we calculated the characteristic polynomial using `Det`. *Mathematica* also has a built-in command to calculate characteristic polynomials [100✱].

```
??CharacteristicPolynomial
```

`CharacteristicPolynomial[`*matrix*`, `*var*`]`
calculates the characteristic polynomial of the square matrix *matrix* in the variable *var*.

Here is a simple example.

```
CharacteristicPolynomial[Table[i + j - 1/j, {i, 3}, {j, 3}], λ]
```

The same result is obtained by calculating $|\mathbf{A} - \lambda\,\mathbf{1}|$.

```
Det[Table[i + j - 1/j, {i, 3}, {j, 3}] - λ IdentityMatrix[3]]
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

6.6 The Top Ten Built-in Commands

As the last application of lists (although a more time-consuming one), we investigate the frequency of use for the nearly 2000 built-in *Mathematica* commands. We conduct our search in the standard *Mathematica* packages.

We begin by finding all built-in commands that we want to count. (We must use `DeleteCases` to get rid of `commands` because it is added to the list of variables before `Names["*`*"]` is carried out.)

```
commands = DeleteCases[Names["*"], "commands"];
```

We now package all commands in `HoldPattern`, which prevents their evaluation, and it is also much more convenient for matching patterns than is the `Unevaluated`, already used on other occasions.

```
allBuiltInNames = (HoldPattern @@ ToHeldExpression[#])& /@ commands;
```

```
Short[allBuiltInNames, 18]
```

Now, we need to load the programs to be searched. We obtain a list of their names using `FileNames`. (We now introduce this command, although, in general, it is not our intention to discuss commands dealing with the operating system.)

`FileNames[`*fileStringNameWithPossibleMetaCharacters*`, `*directory*`, Infinity]`
gives a list of all file names that match *fileStringNameWithPossibleMetaCharacters* and are in the directory *directory* or in any of its subdirectories.

Here are all files of interest to us. (The construction
`Select[$Path, StringMatchQ[#, "*Packages*"]&]`
is needed to access to the package directory and other directories containing `name.m` files, independent of the platform.)

```
files = Union[Flatten[{
filesPackages = FileNames["*.m", #, Infinity]& /@
                    Select[$Path, StringMatchQ[#, "*StandardPackages*"]&],
(* optional *)
filesStartUp = FileNames["*.m", #, Infinity]& /@
                            Select[$Path, StringMatchQ[#, "*StartUp*"]&]}]];
```

```
Short[files, 10]
```

We have the following number of files.

```
Length[files]
```

We now get to the heart of the routine for analyzing the commands: `whichCommandsAreUsed` gives a list whose *i*th element is the number of times the *i*th command in `allBuiltInNames` is used in these packages. In anticipation of its later use, the argument of `whichCommandsAreUsed` should be in the form of a list of *Mathematica* definitions, each enclosed in `Hold`.

We now define a function `hold` such that it meets the following conditions:

■ It is not a built-in function.

■ Its arguments never change.

Thus, we give `hold` only the attribute `HoldAll` and do not give any explicit function definition.

```
SetAttributes[hold, HoldAll];
```

The function `whichCommandsAreUsed` operates as follows. First, all `Hold[Null]` are removed from the list. These `Hold[Null]` were generated while parsing comments and newlines in the packages. Next, all `Hold` commands (built-in commands) at level 1 enclosing the expressions are replaced by the function `hold` defined above. Then, the head `List` of the enclosing list is replaced using `hold`. To get all built-in commands that are used (and to prevent their immediate evaluation), we enclose all atomic expressions with `hold`, using `Map[hold, `*expr*`, {-1}, Heads -> True]`. We split the resulting expression with `Level[hold[…hold[…]…], {-2}, Heads -> True]` into expressions of the form `hold[`*atom*`]`. Finally, using `Count` on these expressions, we count how often each built-in command appears. (A related construction can be found in [194✱], Subsection 5.3.2.)

An alternative approach to using the `HoldAll` attribute and `HoldPattern` is to convert the interesting parts into strings. Then, no danger exists of an evaluation taking place. We believe the implementation given here is more interesting and more elegant.

```
whichCommandsAreUsed[l_?(VectorQ[#, (Head[#] == Hold&)]&)] :=
Module[{buildingBlocks, result},
(* keep current status of spelling messages *)
oldspell = General::spell; oldspell1 = General::spell1;
Off[General::spell]; Off[General::spell1];
buildingBlocks = (* make all hold[...] *)
  Level[Map[hold, hold @@ (Apply[hold, #]& /@  Select[l,
                                   (# =!= Hold[Null]&)]),
             {-1}, Heads -> True],
        {-2}, Heads -> True];
(* now count *)
result = Count[buildingBlocks, #, {-1}]& /@ allBuiltInNames;
(* The last step was simple, but is relatively slow.
   Using hashing and sorting we could considerably speed it up:

  Module[{𝒯 = Table[0, {Length[allBuiltInNames]}], 𝒫},
   MapIndexed[(𝒫[#1] = #2[[1]])&, hold @@@ allBuiltInNames];
   counts = Cases[{𝒫[First[#]], Length[#]}& /@
                        Split[Sort[buildingBlocks]],
              {_Integer, _}];
   Do[𝒯[[counts[[j, 1]]]] = counts[[j, 2]], {j, Length[counts]}];
   result = 𝒯];
*)
(* restore old status of spelling messages *)
If[Head[oldspell]  === String, On[General::spell]];
If[Head[oldspell1] === String, On[General::spell1]];
result]
```

To avoid getting a list of nearly 2000 elements that are mostly 0s (any single package will use only a small fraction of all built-in commands), we define the function `whichCommandsAreUsedWithCommand`. This routine uses the result of `whichCommandsAreUsed` and produces an ordered list of the number of times the built-in commands appear.

```
whichCommandsAreUsedWithCommand[l_List] :=
Sort[ (* more often used commands come first *)
  Select[Thread[(* mix number and names *)
               {commands, l}], (#[[2]] != 0)& ],
       OrderedQ[{#2[[2]], #1[[2]]}]& ]
```

We now test it.

```
a = Append; b = Plot; (* to see if a and b are evaluated *)
whichCommandsAreUsedWithCommand[
whichCommandsAreUsed[{Hold[a; a + 1; a + 2],
                      Hold[2 3],
                      Hold[{6}],
                      Hold[Function[Sin, Sin + Cos]],
                      Hold[b[c]],
                      Hold[hold],
                      Hold[N @@ (r& /@ {s, ss})],
                      Hold[Quit[]],
                      Hold[ReleaseHold[Hold[E]]],
                      Hold[Hold],
                      Hold[N[6]],
                      Hold[1 = 2],
                      Hold[6[N]],
                      Hold[$IterationLimit]}
                     ]]
```

Our implementation worked perfectly. Nothing is evaluated, `Append` and `Plot` do not appear, `$IterationLimit` does appear, and neither `Quit` nor `1 = 2` leads to an error message or quit the kernel. Moreover, the outermost list and the occurrences of `Hold` at level 1 are not counted.

Let us detour for a moment, and let us use the just-implemented functions to analyze which commands have been used how often inside the current notebook. To do this, we read the current notebook as a *Mathematica* expression.

```
thisNotebook = Get[ToFileName["FileName" /.
                  NotebookInformation[EvaluationNotebook[]]]];
```

All inputs appear in cells of type `"Input"`.

```
inputCells = Cases[thisNotebook,
                  Cell[_, "Input" (* | Program *), ___], Infinity];
```

We extract the actual inputs and transform them into held *Mathematica* expressions. (The message `Trace::shdw` comes from the function ``Global`Trace`` introduced in Subsection 6.5.1.)

```
inputCells // Length

heldInputs = DeleteCases[
   Which[Head[#[[1]]] === String, ToHeldExpression[#[[1]]],
         Head[#[[1]]] === TextData,
         (* convert syntactically correct expressions *)
         If[SyntaxQ[#], ToHeldExpression[#]]&[
            (* make input string *)
            StringJoin[#[[1, 1]] /. (* remove style of comments *)
                       StyleBox[s_, ___] :> s]]]& /@ inputCells,
                          Null | $Failed];
```

Here is the result of which functions have been used how often.

```
whichCommandsAreUsedWithCommand[
         whichCommandsAreUsed[heldInputs]] // Take[#, 20]&
```

In case, the reader is wondering about the relatively large number of occurrences of `Null` in the last result: They arise from inputs like the following.

```
FullForm[Hold[a; b;]]
```

Here is the total number of occurrences of built-in functions.

```
Plus @@ (Last /@ %%)
```

Here is the same done with the (larger) Chapter 1 of the Symbolics volume [303✱]. Because we intentionally use some incorrect syntax in this chapter, the building of `heldInputs` uses the `If[SyntaxQ[#],…]` construction.

```
chapterS1Notebook = Get[ToFileName[ReplacePart["FileName" /.
   NotebookInformation[EvaluationNotebook[]], "4_Symbolics_1.nb", 2]]];

(* all input cells of 4_Symbolics_1.nb *)
inputCells = Cases[chapterS1Notebook, Cell[_, "Input", ___], Infinity];

heldInputs = If[SyntaxQ[#], ToHeldExpression[#], Sequence @@ {}]& /@
 DeleteCases[
  Which[Head[#[[1]]] === String, #[[1]],
        Head[#[[1]]] === TextData,
          StringJoin[#[[1, 1]] /. StyleBox[s_, ___] :> s]]& /@ inputCells,
             Null, {1}];
```

Altogether, more than 67000 occurrences of built-in functions exist and these are the most used ones. Carrying out the next input yields this result:
`{67510,{{Times,8190},{List,6615},{Power,5883},{Plus,4421},{Set,2789}}}`.

```
{Plus @@ (Last /@ #), Take[#, 5]}&[
   whichCommandsAreUsedWithCommand[whichCommandsAreUsed[heldInputs]]]
```

It now remains to analyze all standard packages. We cannot do this with `Get`, of course, as this would lead to the immediate evaluation of the *Mathematica* commands contained there. Instead, we use `ReadList`.

`ReadList[`*file*`, Hold[Expression]]`

gives a list of all *Mathematica* expressions in *file*, each enclosed in `Hold`. Comments in *file* of the form `(*` *comment* `*)` yield `Hold[Null]`.

We look at `files`. This is the first package.

```
files[[1]]
```

We read it in.

```
ReadList[files[[1]], Hold[Expression]];
```

This package consists of 30 “lines”.

```
Length[%]
```

We are ready to analyze the first package.

```
Off[General::spell]; Off[General::spell1];
Timing[whichCommandsAreUsedWithCommand[whichCommandsAreUsed[
                        ReadList[files[[1]], Hold[Expression]]]]]
```

Because it would take a relatively large amount of time, we do not run the following input that analyses all packages.

```
res = whichCommandsAreUsedWithCommand[
Sum[whichCommandsAreUsed[ReadList[files[[i]], Hold[Expression]]],
    {i, Length[files]}]]
```

The following result was calculated from all packages in the standard package directory. Its routines contained 170112 appearances of built-in *Mathematica* commands in the context `System``. Here is an ordered list of the most-used commands.

Command	Appearances
List	19836
Pattern	12910
Blank	12743
Set	11940
Times	10392
Power	6354
Plus	5352
Null	5194
CompoundExpression	4848
SetDelayed	3672
⋮	

Because of changes in the packages with each release of *Mathematica*, the reader's results might be different from the just-calculated ones.

The frequency of occurrence of all commands is best visualized graphically. Let $p(k)$ be the frequency ordered decreasingly, and then using the following code we can generate the following log–log plot, showing over a broad region Zipf's law $p(k) = a\,k^{-\rho}$ [156✱], [266✱], [227✱], [307✱], [217✱], [197✱], [121✱], [276✱], [216✱], [65✱], [9✱], [93✱], [124✱], [82✱], [201✱], [300✱], [240✱], [305✱], [306✱], [338✱], [153✱], and [113✱].

```
data = Reverse[Sort[DeleteCases[(#/Plus @@ #)&[(Last /@ res)], 0]]];

(* clean-up of used symbols *)
Needs["Graphics`Graphics`"]

Remove[Global`LogLogListPlot]

Remove /@ ToExpression[StringJoin["Global`", StringDrop[#, 18]]]& /@
                                          Names["Graphics`Graphics`*"];

Needs["Graphics`Graphics`"]

LogLogListPlot[data, Frame -> True, FrameLabel -> {"k", "p(k)"},
               PlotJoined -> True, PlotRange -> All,
               PlotStyle -> {Hue[0], Thickness[0.004]}]
```

It may also be of interest to look at the commands in last place. Some of the built-in functions appear in none of the packages. We leave it to the user to investigate which ones. But they are needed anyway. The user will probably make use of some of these commands from time to time, mostly in interactive work rather than using them in packages. They often deal with “fine-tuning” graphics and with numerical routines.

Now, let us analyze the *Mathematica* source code from this book. Using the input from above for the analysis of Chapter 1 of the Symbolics volume [303✱] for all chapters, we can determine which *Mathematica* functions were used how often. In summary, the source code contains about 435000 occurrences of built-in commands. These are my top ten.

```
guideBooksChapterFileNames = ToFileName[ReplacePart["FileName" /.
  NotebookInformation[EvaluationNotebook[]], #, 2]]& /@
     {"1_Programming_1.nb", "1_Programming_2.nb", "1_Programming_3.nb",
      "1_Programming_4.nb", "1_Programming_5.nb", "1_Programming_6.nb",
      "2_Graphics_1.nb", "2_Graphics_2.nb", "2_Graphics_3.nb",
      "3_Numerics_1.nb", "3_Numerics_2.nb",
      "4_Symbolics_1.nb", "4_Symbolics_2.nb", "4_Symbolics_3.nb"};
```

```
Off[Syntax::com]; Off[Precision::precsm];
allHeldInputs = Module[{aux},
Table[(* read in the notebook *)
   nb = Get[guideBooksChapterFileNames[[i]]];
   (* analyze the notebook *)
   inputCells = Cases[nb, Cell[_, "Input" | "Program", ___], Infinity];
   heldInputs = If[SyntaxQ[#], ToHeldExpression[#], Sequence @@ {}]& /@
   DeleteCases[Which[Head[#[[1]]] === String, #[[1]],
                     Head[#[[1]]] === TextData,
                     aux = #[[1, 1]] /. StyleBox[s_, ___] :> s;
                     If[Head[aux] === String ||
                        Union[Head /@ aux] === {String},
                        StringJoin[aux]]]& /@ inputCells,
               Null, {1}], {i, 14}] // Flatten];

res = whichCommandsAreUsedWithCommand[whichCommandsAreUsed[allHeldInputs]];

Plus @@ (Last /@ res)

Take[res, 12]
```

Rank	Command	Appearances
1	`List`	57555
2	`Times`	43986
3	`Power`	29205
4	`Plus`	24639
5	`Slot`	19131
6	`Set`	18396
7	`Blank`	15862
8	`Pattern`	15227
9	`Rule`	15116
10	`CompoundExpression`	13375

The relative frequency of all commands is now given by the following picture.

```
data = Reverse[Sort[DeleteCases[(#/Plus @@ #)&[(Last /@ res)], 0]]];

Needs["Graphics`Graphics`"]

LogLogListPlot[data, Frame -> True, FrameLabel -> {"k", "p(k)"},
               PlotJoined -> True, PlotRange -> All,
               PlotStyle -> {Hue[0], Thickness[0.004]}]
```

We could, of course, also extract all evaluatable cells from the 14 chapter notebooks and evaluate them in a new notebook. The following code extracts the 21000+ evaluatable cells.

```
allEvaluatableCells = DeleteCases[Flatten[
Table[Cases[(* read in chapter notebook *)
            Get[guideBooksChapterFileNames[[k]]],
            (* extract evaluatable cells *)
            Cell[_, "Input" | "Program" | "StandardFormInput", ___],
            Infinity] //.  (* make evaluatable input *) "Program" -> "Input" ,
       {k, 14}]], Cell[___, Evaluatable -> False, ___]];

Length[allEvaluatableCells]
```

Evaluating all cells would at once would result in a lot of problems (interfering variable names, unreasonable large memory demand, etc.). To avoid such problems, we could define a `$Pre`-function that avoids the actual evaluation of each input.

```
(* to avoid any actual evaluation; just parsing is enough *)
SetAttributes[hold, HoldAll];
(* to later get out of the hold-mode *)
EscapeTheHold /: hold[EscapeTheHold] := Unset[$Pre]

(* create a new notebook with the inputs *)
nbAllInputs = NotebookPut @ Notebook[
Flatten[{(* to avoid full evaluation of the inputs *)
         Cell["$Pre = hold", "Input"],
         (* avoid spelling annoying messages *)
         Cell["Off[General::spell]; Off[General::spell1];", "Input"],
         Take[allEvaluatableCells, (* maybe less *) All]}]];

LineNumberBefore = $Line;
(* select and evaluate all cells—this will take some hours *)
FrontEndTokenExecute[nbAllInputs, "SelectAll"];
FrontEndTokenExecute[nbAllInputs, "EvaluateCells"];
```

Removing now the paralyzing `$Pre` with the above setup `EscapeTheHold` will bring *Mathematica* back to a state where we can fully evaluate inputs. Now all the evaluatable inputs of the four *GuideBooks* are stored in the downvalues of `In`. Extracting them gives the expression `allInputExpressions` containing held versions of all input. The interested reader can continue to carry out statistics on these inputs (such as `ByteCount`, `LeafCount`, …), but we end here.

```
(* de-paralyze Mathematica *)
EscapeTheHold
LineNumberAfter = $Line;

(* extract and freeze inputs from the GuideBooks *)
allInputExpressions = Extract[#, 2, Hold]& /@
  Take[DownValues[In], {LineNumberBefore + 3, LineNumberAfter - 2}]
```

We could go on with related investigations, for instance, with the question: When writing a *Mathematica* package, which keys are most often pressed? Let us calculate a detailed result of analyzing all packages in this respect.

We again make use of the command `ReadList`. In the form `ReadList[`*file*`, Record, RecordSeparators -> {}]`, a file is read in as one string. We then divide it into its building blocks and count the frequency of their appearances. (This means we also count the not typed notebook structures like `Cell`.) Here is the corresponding program. (We use the same files as above.)

```
Module[{char, allOccuringCharacters, all},
Do[char[i] = Sort[
   Function[allLetters,(* count letters *)
   {{#, Count[allLetters, #]}& /@ Union[allLetters]}][
   Characters[StringJoin @@ Flatten[
   (* read in the file *)
   ReadList[files[[i]], Record, RecordSeparators -> {}]]]][[1]],
    OrderedQ[{#2[[2]], #1[[2]]}]&], {i, 1, Length[files]}];
   (* add results for all files *)
   allOccuringCharacters = Union[
    First /@ Flatten[Table[char[i], {i, 1, Length[files]}], 1]];
   all = Flatten[Table[char[i], {i, 1, Length[files]}], 1];
   result = {Union[First[#]], Plus @@ Last[#]}&[
               Transpose[Cases[all, {#, _}]]]& /@ allOccuringCharacters];
```

We do not execute this program. Here is the result of executing it.

```
InputForm[Take[result = Sort[{#1[[1]], #2}& @@@ result,
                              OrderedQ[{#2[[2]], #1[[2]]}]&], 26]]
```

This results in a total of about three million characters for the packages. The following input calculates the exact result.

```
Plus @@ (Last /@ result)
```

For characters, the following form of Zipf's law holds approximately: $p(k) = \alpha - \beta \ln(k)$, where it is the occurrence probability for the letter k, and the probabilities are sorted.

```
With[{data = Last /@ result},
ListPlot[MapIndexed[{Log[#2[[1]]], #1}&,
                    (Last /@ data)/Plus @@ (Last /@ data)],
         PlotRange -> {0, 0.075}, PlotJoined -> True,
         PlotStyle -> {Hue[0], Thickness[0.004]}]]
```

Now, once we have read in all files, we could go on and answer related questions of interest. So, how deeply are *Mathematica* programs nested? The following program counts this for all definitions from files. The output is in the form *depthOfRoutine, numberOfSuchRoutines*.

```
Off[General::spell1]; Off[Read::readt];

Function[arg, {#, Count[arg, #]}& /@ Union[arg]][
   Flatten[Array[Depth /@ DeleteCases[ReadList[files[[#]],
              Hold[Expression]], Hold[Null]]&, Length[files]]]] // TableForm
```

The result is as follows.

Depth	**Number**
2	35
3	1234
4	3533
5	1210
6	1382
7	1619
8	1016
9	806
	…
30	3
32	1
34	1
35	1

(Following the last investigation, we could now analyze how many functions a given functions directly calls [322✱].)

We could try to analyze the file system of the computer from inside *Mathematica*, for instance, with the following input.

```
FixedPoint[Map[If[FileType[#] === Directory, FileNames["*", {#}], #]&,
               #, {-1}]&,
           Flatten[FileNames["*", {#}]& /@
                       {FixedPoint[ParentDirectory, Directory[]]}]]
```

But, as mentioned in the preface, we will not discuss file-related things in this book, and so, we do not go into the details of these commands.

We now could go on and investigate some statistical properties of the notebooks forming this book. Because the following operations are quite memory intensive, we do not carry them out here by default. I recommend restarting *Mathematica* if the reader wants to run the following inputs. I also recommend using the unevaluated notebooks to avoid reading very large notebooks into the *Mathematica* kernel. To avoid reading in large amounts of PostScript graphics and *Mathematica* outputs, the following inputs are best run on notebooks that have all graphics and outputs removed. The reader might get different results when running the following input, because of a different number of output cells currently present, modified input cells, …. In the following inputs, we frequently turn off messages. While various messages could be avoided by using a more careful programming, to avoid the presentation of many long programs we will not do this here.

How many styles are used how often?

```
cellData = Table[(* read in the notebook *)
   nb = Get[guideBooksChapterFileNames[[i]]];
   (* analyze the notebook *)
   allCells = Select[Cases[nb, _Cell, Infinity], Length[#] > 1& ];
   {#[[1]], Length[#]}& /@ Split[Sort[#[[2]]& /@ allCells]], {i, 14}];

(* add all data together *)
Sort[Function[cs, {cs, Plus @@ (* extract style data *)
            (Last /@ Cases[Flatten[cellData, 1], {cs, _}])}] /@
                          Union[Flatten[Map[First, cellData, {2}]]],
     #1[[2]] > #2[[2]]&] // TableForm
```

Here are the first ten entries from the last result:

```
Input                20910
Text                 19089
BibliographyItem      9535
InlineFormula         5696
DisplayFormula        1362
SolutionSubgroup       979
MathDescription        941
TextDescription        871
DescriptionTop         621
DescriptionBottom      621
```

Now, we could investigate typeset formulas. Which boxes are used, and how often do they appear?

```
(* the box types we are looking for *)
boxTypes = _ButtonBox | _CounterBox | _ErrorBox | _FormBox | _FractionBox |
           _FrameBox | _GridBox | _OverscriptBox | _RadicalBox | _RowBox |
           _SqrtBox | _StyleBox | _SubscriptBox | _SubsuperscriptBox |
           _SuperscriptBox | _TagBox |  _UnderscriptBox;

boxData = Table[(* read in the notebook *)
                nb = Get[guideBooksChapterFileNames[[i]]];
                (* analyze the notebook *)
               allBoxes = Head /@ Cases[nb, boxTypes, Infinity];
               {#[[1]], Length[#]}& /@ Split[Sort[allBoxes]], {i, 14}];

(* add all data together *)
Sort[Function[cs, {cs, Plus @@ (* extract box data *)
            (Last /@ Cases[Flatten[boxData, 1], {cs, _}])}] /@
                         Union[Flatten[Map[First, allBoxes, {2}]]],
     #1[[2]] > #2[[2]]&] // TableForm
```

The first 10 entries in the result are:

```
StyleBox             61 067
RowBox               60 476
FormBox              21 597
ButtonBox            19 427
CounterBox           17 640
SubscriptBox         12 986
SuperscriptBox       10 315
FractionBox           2578
SubsuperscriptBox     1930
TagBox                 895
```

Which options are used in the notebooks, and how often do they appear?

```
optionsData =
Table[(* read in the notebook *)
      nb = Get[guideBooksChapterFileNames[[i]]];
      (* analyze the notebook *)
      allOptions = First /@ Cases[nb, _Rule, Infinity];
      {#[[1]], Length[#]}& /@ Split[Sort[allOptions]], {i, 14}];

(* add all data together *)
Select[Sort[Function[cs, {cs, Plus @@ (* extract option data *)
            (Last /@ Cases[Flatten[optionsData, 1], {cs, _}])}] /@
                        Union[Flatten[Map[First, optionsData, {2}]]],
       #1[[2]] > #2[[2]]&],
      (* take only the most frequent ones *) #[[2]] > 5&] // TableForm
```

These are the ten most-used options.

```
ButtonStyle          21 931
CellTags             15 639
FontSlant             2927
FontWeight            1268
ParagraphSpacing       697
Editable               568
LimitsPositioning      367
ScriptLevel            287
Evaluatable            263
MultilineFunction      238
```

What is the ratio between text and *Mathematica* input in this book? The following two graphics try to answer this question. The left graphic shows the running ratio between the number of input cells and the number of text cells. The right graphic shows the running ratio between the `ByteCount` of the input cells and the `ByteCount` of the text cells. Each chapter is represented as one line, ranging from red (Chapter 1 of the Programming volume) to dark blue (Chapter 3 of the Symbolics volume). Here are a few observations from these plots:

■ In the beginning, the ratios are small, meaning that text cells dominate the beginnings of the chapters

■ The longest is Chapter 1 of the Symbolics volume, followed by Chapter 1 of the Numerics volume.

■ The two chapters with the most *Mathematica* inputs (having a `ByteCount` ratio of about 1) are the two graphics Chapters, 1 and 2.

■ Asymptotically, the ratio between input cells and text cells is about 1 for all chapters, meaning that each *Mathematica* input has some kind of corresponding text cell.

■ The large jump in the `ByteCount` ratio for Chapter 1 of the Symbolics volume is caused by the large implicit representation of the trefoil knot from Subsection 1.9.3 of the Symbolics volume [303✱].

```
Off[General::dbyz]
cellTypeData = Module[{cells},
Table[(* read in the notebook *)
      nb = Get[guideBooksChapterFileNames[[i]]];
      (* extract input- and text-cells *)
      cells = Cases[nb, Cell[___, "Input", ___] |
                        Cell[___, "Text", ___], Infinity];
      (* count input- and text-cells *)
      Apply[Divide, Transpose[Rest[FoldList[Plus, 0,
                     If[MatchQ[#, Cell[___, "Input", ___]],
            {{1, 0}, {ByteCount[#], 0}},
            {{0, 1}, {0, ByteCount[#]}}]& /@ cells]]], {-2}], {i, 14}]];

Show[GraphicsArray[
Function[fl, Graphics[
MapIndexed[{Hue[(#2[[1]] - 1)/18], #1}&,
     Line /@ (MapIndexed[{#2[[1]], #1}&, fl[#]]& /@ cellTypeData)],
                PlotRange -> All, Frame -> True]] /@ {First, Last}]]
```

How deep are notebooks structured? We count the number of expressions at level *i* as a function of *i*. We display the result as a graphic.

```
Off[ReplaceAll::reps]; Off[StringReplacePart::string];
Off[Get::string]; Off[ToFileName::strse]; Off[Part::partd];
depthData = Table[(* make notebook filename *)
   (* read in the notebook *)
   nb = Get[guideBooksChapterFileNames[[i]]];
   (* analyze the notebook *)
   Table[{k, Length[Level[nb, {k - 1}]]}, {k, Depth[nb]}], {i, 14}];

ListPlot[(* add all data together *)
Sort[Function[cs, {cs, Plus @@ (* extract depth data *)
            (Last /@ Cases[Flatten[depthData, 1], {cs, _}])}] /@
                           Union[Flatten[Map[First, depthData, {2}]]],
     #1[[2]] > #2[[2]]&], PlotRange -> All,
         PlotStyle -> {Hue[0], PointSize[0.008]}]
```

We could also analyze some more content-related issues. How many references are used, and from which year do they come?

```
(* extract the part of the bibliography cell that contains the year *)
getYearString[bibliographyItemCell_] :=
Module[{textData = bibliographyItemCell[[1, 1]] //.
          {a___, s1_String, s2_String, b___} :> {a, s1 <> s2, b}},
        If[# =!= {}, Last[#], {}]&[
        DeleteCases[Cases[textData, _String],
                    _?(Union[Characters[#]] === {" "} ||
                       Union[Characters[#]] === {"."}&)]]]

(* extract the year from the last characters of a bibliography item *)
getYear[s_String] :=
With[{sp = StringPosition[s, "."][[-1, -1]] - 1},
      If[(* journal or book? *)
         StringTake[s, {sp, sp}] === ")", StringTake[s, {sp - 4, sp - 1}],
         StringTake[s, {sp - 3, sp}]]];

getYear[{}] := Sequence[]

(* extract bibliographic data *)
bibliographyData =
Table[(* read in the notebook *)
      nb = Get[guideBooksChapterFileNames[[i]]];
      (* analyze the notebook *)
      bibliographyCells = Cases[nb, Cell[_, "BibliographyItem", _],
                                Infinity];
      Select[getYear[getYearString[#]]& /@ bibliographyCells,
             If[SyntaxQ[#], 1800 < ToExpression[#] <= 2004]&], {i, 14}];

(* add all data together *)
(allBibliographyData = Sort[{ToExpression[#[[1]]], Length[#]}& /@
                            Split[Sort[Flatten[bibliographyData]],
                                  #1[[2]] > #2[[2]]&]);

(* earliest and latest references *)
{Take[#, +15], "<<" <> ToString[Length[#] - 20] <> ">>",
 Take[#, -15]}&[allBibliographyData]

(* visualize results *)
Show[GraphicsArray[
Block[{$DisplayFunction = Identity,
       opts = Sequence[Axes -> False, Frame -> True, PlotRange -> All,
                       PlotStyle -> {Hue[0], PointSize[0.008]}]},
{(* plot the data *)
 ListPlot[N[allBibliographyData], opts],
 (* logarithmic plot of the data *)
 ListPlot[{#[[1]], Log[#[[2]]]}& /@ N[allBibliographyData], opts]}]]]
```

This is the result. The picture shows the author's effort to keep the references up to date. The second plot is the logarithmic one. We skip the Zipf law for the references [277✱].

How many (different) words appear in the text (in `Text`-style cells)? (For a computational analysis of the English language in general, see [178✱].)

```
textData =
Table[(* read in the notebook *)
      nb = Get[guideBooksChapterFileNames[[i]]];
      (* analyze the notebook *)
       texts = Join[ (* take out the pure text parts *)
          Cases[DeleteCases[Cases[
             (* remove non-Text cells *)
             DeleteCases[nb, Cell[_TextData, "Input", ___] |
                             Cell[_TextData, "Program", ___] |
                             Cell[_TextData, "BibliographyItem", ___],
                        (* keep italicized words *)
                        Infinity] /. StyleBox[it_, "TI"] :> it,
                TextData[___], Infinity],
                 (* do not take inputs, hyperlinks, references, … *)
                      _StyleBox | _BoxData | _Cell |
                      _CounterBox | _ButtonBox, Infinity],
               _String, Infinity],
             First /@ Cases[nb, Cell[_String, "Text", ___], Infinity]],
       {i, 1, 14}];

(* split the string part into single words *)
splitString1[s_String] := StringTake[s, #]& /@ ({1, -1} + #& /@
    Partition[Flatten[{0, StringPosition[s,
    (* dividing characters *) {", ", ". ", ": "}],
                      StringLength[s] + 1}], 2])

splitString2[s_String] :=
    With[{l = StringLength[s]},
    StringTake[s, #]& /@ ({1, -1} + #& /@
          Partition[Flatten[{0, DeleteCases[
                  StringPosition[s, {" ", "—", "-"}], {l, l}], l + 1}],
                            2])]

(* separate out all single words *)
extractWords = (Select[ToLowerCase /@ DeleteCases[Flatten[splitString2 /@
                              Flatten[splitString1 /@ #]], ""], LetterQ] /.
                                         "mathematica" -> "Mathematica")&;

(* words used in the 14 chapters *)
allUsedWordsList = extractWords /@ textData;

wordCountData = {Length[Union[#]], Length[#]}& /@ allUsedWordsList;
```

After running the above code, we get the following results. The text of the *GuideBooks* has about 450000 words and about 8300 different words.

```
(* number of words and number of different words for all chapters *)
{Length[allUsedWords = Flatten[allUsedWordsList]],
 Length[differentUsedWords = Union[allUsedWords]]}
```

The number of different words used in a chapter nn compared to the total number of words $\mathcal{N}$ defines the lexical wealth $k = n / N$. A power law is conjectures to hold in the form $\mathcal{N} = \alpha\, k^{\beta}$ [110✱].

```
Show[GraphicsArray[
Block[{$DisplayFunction = Identity,  opts =
       Sequence[PlotRange -> All, Frame -> True, Axes -> False]},
  {(* show 𝒩 ~ n^γ *)
   ListPlot[Log[wordCountData], opts],
   (* show 𝒩 ~ (n/𝒩)^β *)
   ListPlot[{Log[#1/#2], Log[#2]}& @@@ wordCountData, opts]}]]]

(* fit the power law *)
Exp[Fit[{Log[#1/#2], Log[#2]}& @@@ wordCountData, {1, k}, k]] // Together
```

Here are the most frequently used words.

```
wordStatistics = Sort[{#[[1]], Length[#]}& /@ Split[Sort[allUsedWords]],
                       Last[#1] > Last[#2]&];

GridBox[Take[wordStatistics, 20],
        ColumnAlignments -> {Left, Right}] //  DisplayForm
```

Comparing the ranked words from different chapters shows a high degree of coinciding words [332✱]. Because the chapters were written largely in parallel, this is to be expected.

```
(* words in the chapters *)
allUsedWordsInChapters = extractWords /@ textData;

(* ranked words in selected chapters *)
rankedUsedWordsInChapters =
Map[#[[1]]&, Take[Sort[{#[[1]], Length[#]}& /@
      Split[Sort[allUsedWordsInChapters[[#]]]],
            Last[#1] > Last[#2]&], 15]]& /@ {6, 7, 8, 10, 12, 14};

(* show ranked words in selected chapters *)
TableForm[Transpose[rankedUsedWordsInChapters],
   TableHeadings -> Map[StyleForm[#, FontWeight -> "Bold"]&,
         {Range[15], {"P6", "G1", "G2", "N1", "S1", "S3"}}, {-1}],
          TableSpacing -> {0.2, 2}]
```

Let us check how often typical mathematics book words appear (for a top-ten list of mathematics article titles, see http://www.maths.leeds.ac.uk/~pmt6jrp/personal/mathswords.html).

```
Cases[wordStatistics, {"integrate", _} | {"differentiate", _} |
                      {"solve", _} | {"sum", _} | {"multiply", _} |
                      {"derive", _} | {"substitute", _} | {"vanish", _}]
```

```
Cases[wordStatistics, {"trivial", _} |
                      {"straightforward" | "straightforwardly", _} |
                      {"easily", _} | {(* left to the *)"reader", _}]
```

How often was *Mathematica* mentioned throughout all 14 main chapters?

```
Plus @@ (Count[Get[#], "Mathematica", {-1}]& /@ guideBooksChapterFileNames)
```

The frequency of occurrence of all words is again best visualized graphically.

```
Needs["Graphics`Graphics`"]

LogLogListPlot[Reverse[Sort[(#/Plus @@ #)&[Last /@ wordStatistics]]],
               Frame -> True, FrameLabel -> {"k", "p(k)"},
               PlotJoined -> True, PlotRange -> All,
               PlotStyle -> {Hue[0], Thickness[0.004]}]
```

Having counted the words, it is easy to calculate the frequency of the various letters inside the more than two million characters.

```
Take[Sort[{#[[1]], Length[#]}& /@
        Split[Sort[Flatten[Characters /@ allUsedWords]]],
                   Last[#1] > Last[#2]&], 26]
```

As in Subsection 6.4.2, we could investigate still more questions, such as the average number of inputs between two text cells, statistics about the size of the exercise solutions, the number of comments (* … *) in the *Mathematica* inputs, the distribution of *Mathematica* commands in the notebooks themselves (viewed as *Mathematica* expressions), the number of diagonal links [289✱], the connectivity of the referenced papers [225✱], [226✱], the connectivity and clustering properties of *Mathematica* code as a network [7✱], [76✱] (considering say, the built-in functions as elements and the appearance as an argument as an edge) …, but we will end here and leave it to the reader to continue this kind of investigations. In Chapter 1 of the Numerics volume [302✱], we will return to some related considerations—we will analyze long-range correlations in Shakespeare's *Hamlet*.

Σ (* session summary *) **`TMGBs`PrintSessionSummary[]`**

Overview

```
Get[ToFileName[ReplacePart[
            "FileName" /. NotebookInformation[EvaluationNotebook[]],
            "ChapterOverview.m", 2]]];

ChapterOverview["Programming", 6]
```

Exercises

1.^{L2} Benford's Rule

Given a long list of empirical data (e.g., lengths of rivers, areas of deserts and seas, addresses, bank account balances, physical data, chemical data), check whether this data satisfies Benford's rule: The probability distribution of the appearance of the digit i ($1 \le i \le 9$) in the first place in a data entry is $\log_{10}(1 + 1/i)$, where all zeros to the left of the first significant digit are ignored. For details on Benford's rule, see [139✱], [137✱], [337✱], [32✱], [242✱], [94✱], [265✱], [241✱], [243✱], [249✱], [61✱], [71✱], [186✱], [51✱], [50✱], [66✱], [138✱], [153✱], [164✱], [252✱], and [228✱] and the references therein.

2.^{L1} `Map`, `Outer`, `Inner`, and `Thread` versus `Table` and `Part`, Iteratorless Generated Tables, Sum-free Sets

a) Compare the computational times for `Map`, `Outer`, `Inner`, and `Thread` on reasonably-sized vectors to those for `Table`, `Do`, and `Part` using analogous constructions.

b) Write a function that generates the same output as
`Table[`f`[`i_1`, `i_2`, …, `i_n`], {`i_1`, 1, …, `m`}, {`i_2`, 1, …, `m`}, …, {`i_n`, 1, …, `m`}]`
but which does not contain any explicit iterator variable.

c) Given a square matrix **A** of dimension d with elements a_{ij} and a vector of operators **f** of length d with elements f_j, form a new matrix **B** with elements $b_{ij} = f_j(a_{ij})$. Write several different programs that form the matrix **B**, and compare their timings for some matrices **A** and vectors **f** of different dimensions.

d) Given a set $\mathcal{S}_o$ of positive integers $\{n_1, \ldots, n_o\}$, recursively enlarge this set by adding the smallest integer that cannot be expressed as a sum $n_i + n_j$, $1 \le i, j \le o$ [86✱].

Implement the recursive enlargement first using a procedural (list-based) approach and then using a caching approach. Compare the timing of the two approaches for the starting set of the first ten primes for 2000 recursive enlargements.

3.^{L1} Index

Create an index for the *Mathematica* commands that are introduced in this book. It should consist of a list of the form `{…, {"`*command*$_i$`", "`*chapterSectionSubsection*$_i$`"}, …}`. The function `whereIntroduced[`*command*`]` should give the number of the section where the command is introduced. Check that no command was misspelled when it was introduced. Which commands were introduced twice? The list of commands can be found in the package `Chapter`\
`Overview``.

4.^{L3} Functions Used Too Early?, Check of References, Closing `]]`, Line Lengths, Distribution of Initials, Check of Spacings

a) In the preface, we stated our aim that every time a command is used in this book, it should already have been discussed. Create a *Mathematica* program to check for specific examples to see how close we came to our goal. Collect all commands that have been introduced in gray boxes in a list `alreadyIntroducedCommands`. The command `$Pre` might be useful here.

b) This *GuideBooks* have many references. Check if each mentioned reference is really present and if each reference is at least mentioned once. Which journals are the most cited? What is the number of electronic papers referenced and how did the fraction of electronic papers change over the last years?

c) What are the most common first letters of the initials and last names of all authors of the quoted papers and books?

d) What is the distribution of the line lengths used in the inputs of this book? How much white space (in the form of raw space characters) is on average present in the inputs? What is the average density of code comments?

e) Brackets are very prominent in *Mathematica* code. Not more than two opening brackets can occur in a row, but arbitrarily many closing brackets can occur in a row. Analyze the literal inputs of the *Mathematica GuideBooks* to determine how often n closing brackets occur. If the inputs had been expressed in `FullForm`, how often would n closing brackets occur?

f) As discussed in the Introduction, the inputs of the *GuideBooks* are in `InputForm`. To make the inputs as easy as possible to read, care has been taken to format them properly. This includes white spaces after all commas, white spaces around operators with relatively low binding power (such as `->` or `/.`). Write a program that checks for violations of such spacing rules and check all inputs from this volume of the *GuideBooks*.

g) If one considers language as a network and the words as vertices, one can analyze the distribution of neighbors in this network. The natural interpretations of neighbors of a word are the preceding and postceding words. (Viewing sentences as natural units of a language, the first word of a sentence does not have a precessor and the last word does not have a postceder [234✱].) Analyzing large amounts of sentences and graphing the resulting (binned) frequency of the number of neighbors versus the number of neighbors in a double logarithmic plot shows two clearly different linear regimes [76✱]. Analyze the texts of the *GuideBooks* and see if, despite this relatively small amount of data and the nonnativeness of the author, the two different power laws are nevertheless present.

5.^{L1} Tube Points

Write two different programs to solve the following problem. Suppose we are given lists of the form

$$
\begin{aligned}
&\texttt{points} = \{p_1,\ p_2,\ p_3,\ \ldots,\ p_n\}\\
&\texttt{radii} \;\;= \{r_1,\ r_2,\ r_3,\ \ldots,\ r_n\}\\
&\texttt{vecv} \;\;\;= \{v_1,\ v_2,\ p_3,\ \ldots,\ v_n\}\\
&\texttt{vecu} \;\;\;= \{u_1,\ u_2,\ u_3,\ \ldots,\ u_n\}\\
&\texttt{cos} \;\;\;\;= \{c_1,\ c_2,\ c_3,\ \ldots,\ c_m\}\\
&\texttt{sin} \;\;\;\;= \{s_1,\ s_2,\ s_3,\ \ldots,\ s_m\}
\end{aligned}
$$

Here, p_i, v_i, u_i are vectors of the form $\{px_i,\ py_i,\ pz_i\}$, $\{vx_i,\ vy_i,\ vz_i\}$, $\{ux_i,\ uy_i,\ uz_i\}$. The px_i,… are atomic objects (in a typical application, real numbers); the c_i, s_i, r_i are assumed to be atomic objects.

Create a list of the following form:

$$
\begin{aligned}
\{\{&p_1 + r_1\, c_1\, v_1 + r_1\, s_1\, u_1,\\
&p_1 + r_1\, c_2\, v_1 + r_1\, s_2\, u_1,\\
&p_1 + r_1\, c_3\, v_1 + r_1\, s_3\, u_1,\ \ldots,\\
&p_1 + r_1\, c_m\, v_1 + r_1\, s_m\, u_1\},\\
\{&p_2 + r_2\, c_1\, v_2 + r_2\, s_1\, u_2,\\
&p_2 + r_2\, c_2\, v_2 + r_2\, s_2\, u_2,\\
&p_2 + r_2\, c_3\, v_2 + r_2\, s_3\, u_2,\ \ldots,\\
&p_2 + r_2\, c_m\, v_2 + r_2\, s_m\, u_2\},\\
&\vdots\\
\{&p_n + r_n\, c_1\, v_n + r_n\, s_1\, u_n,
\end{aligned}
$$

$p_n + r_n\ c_2\ v_n + r_n\ s_2\ u_n$,
$p_n + r_n\ c_3\ v_n + r_n\ s_3\ u_n$, …,
$p_n + r_n\ c_m\ v_n + r_n\ s_m\ u_n$}}.

($p_i + r_i\, c_j\, v_i + r_i\, s_j\, u_i$ is a list (head `List`) with three elements.)

6.^{L1} All Subsets

Explain the operation of the following command `allSubsets[`*list*`]`, which produces all subsets of a given set *list*, including the empty set and the set *list* itself. Here is the implementation (coming from [310✱]):

```
allSubsets[l_List] :=
Sort[Distribute[{{}, {#}}& /@ Union[l], List, List, List, Union]]
```

Use such a function definition to implement a one-liner for the sums

$$\mathcal{A}(k_1, k_2, \ldots, k_n) = \frac{1}{K} \sum_{j=0}^{K-1} \prod_{m=1}^{n} \left\lfloor \frac{k_m\, j}{K} \right\rfloor$$

where $K = \prod_{j=1}^{n} k_j$. This sum can be expressed as [291✱]

$$\mathcal{A}(k_1, k_2, \ldots, k_n) = \prod_{j=1}^{n} \left(k_j - 1\right) + \sum_{\{k_{i_1}, \ldots, k_{i_m}\}} (-1)^m \sum_{j=0}^{(k_{i_1}, \ldots, k_{i_m})-1} \prod_{h=i_{m+1}}^{n} \left\lfloor \frac{k_h\, j}{\left(k_{i_1}, \ldots, k_{i_m}\right)} \right\rfloor.$$

Here the outer sum runs over all nonempty subsets of the set $\{k_1, k_2, \ldots, k_n\}$, and the inner product over all k_j not in a given subset. $\left(k_{i_1}, \ldots, k_{i_m}\right)$ denotes the greatest common divisor of the numbers $k_{i_1}, k_{i_2}, \ldots, k_{i_m}$. Calculate $\mathcal{A}(p_1, p_2, \ldots, p_{10})$ where p_k is the kth prime.

7.^{L1} Moessner's Process, Ducci's Iterations

a) Write all integers in natural order. Then delete every second one, and form the following sequence of partial sums:

```
start sequence                1  2  3  4  5  6  7  8   …
delete every second element   1     3     5     7      …
form new partial sums         1     4     9     16     … .
```

They are all squares. Now, delete every third number of the initial sequence, and form the partial sums. If we then strike every second number and again form the partial sums, we get a sequence of cubes.

```
start sequence                1  2  3  4  5   6  7   8  9  …
delete every  third element   1  2     4  5      7   8     …
form new partial sums         1  3     7  12     19  27    …
delete every second element   1        7         19        …
form new partial sums         1        8         27        …
```

Find a functional program for this process that first deletes every ith number and then produces j numbers. Conjecture if $i = 4$ and $i = 5$ lead to fourth and fifth powers?

b) Take four positive integers m, n, p, and q, and form four new integers $|m - n|$, $|n - p|$, $|p - q|$, and $|q - m|$. Iterate this process until it converges [190✱], [141✱], [212✱], [154✱], [46✱].

8.^{L1} Triangles, Group Elements, Partitions, Stieltjes Iterations

Describe what the following pieces of code do.

a)

```
NestedTriangles[n_Integer?Positive] :=
(Function[{x, y}, x.#& /@ y] @@ #)& /@
   Distribute[{Table[{{ Cos[i Pi/2], Sin[i Pi/2]},
                      {-Sin[i Pi/2], Cos[i Pi/2]}}, {i, 0, 3}],
               Flatten[NestList[#/2&,
                           {{{1, 1}, {3, +1}, {1, 3}},
                            {{1, 0}, {2, -1}, {2, 1}}}, n], 1]}, List]
```

Look at the output graphically using the following input.

```
Show[Graphics[Polygon /@ Triangles[6]],
     AspectRatio -> Automatic, PlotRange -> All]
```

b)

```
FixedPoint[Union[Flatten[Outer[Function[C, #]& @
                        Simplify[#1[#2[C]]]&, #, #]]]&,
                 {Function[C, -C], Function[C, (C + I)/(C - I)]}]
```

c)

```
PartitionsLists[n_Integer?Positive] := Drop[FixedPointList[
 Complement[Union[Flatten[ReplaceList[#,
  {{a___, b_, c_, d___} :> {a, b - 1, c + 1, d} /; b - c >= 2,
  {a___, b_, c:(d_ ...), e_, f___} :> {a, b - 1, c, e + 1, f} /;
          b - 1 == d == e + 1}]& /@ #, 1]], #]&,
                      {{n, ##}& @@ Table[0, {n - 1}]}], -2]
```

d)

```
Unprotect[Table];

Table[body_, iters__, Heads -> l_List] :=
With[{d = Length[{iters}]},
Fold[Apply[First[#2], #1, {Last[#2]}]&, Table[body, iters],
         Reverse[MapIndexed[{#1, #2[[1]] - 1}&,
         Take[Flatten[Table[l, {d}]], d]]]]]

Table[body_, iters__, Heads -> l_] := Table[body, iters, Heads -> {l}]
```

e)

```
𝒮𝒜[𝕝_List] := With[{λ = Length[𝕝]},
 Module[{𝕡 = NestList[Flatten[
   Outer[Join, {#}, List /@ Range[Last[#] + 1, λ], 1]& /@ #, 2]&,
             List /@ Range[λ], λ - 1]},
   FixedPointList[Function[ℓ,
    Divide @@@ Partition[Append[Reverse[Apply[(Plus[##])/Length[{##}]&,
     Apply[Times, Map[ℓ[[##]]&, 𝕡, {-2}], {2}], {1}]], 1], 2, 1]], 𝕝]]] /;
                (Or @@ (InexactNumberQ /@ 𝕝)) && (And @@ (NumericQ /@ 𝕝))
```

f)

```
pseudoRandomTree[kStart_] :=
Module[{𝕣, k, 𝓎, 𝕥},
 𝕣 := If[IntegerPart[Abs[Sqrt[2] Sin[Pi k Sin[k = k + 1]]]] === 0,
        0, 2];
 k = kStart; 𝓎[_] := -1;
 𝕥 /: Line[{x_, y_}, 𝕥[]] :=
          Table[{Line[{x, y}, {x + 1, 𝓎[x + 1] = 𝓎[x + 1] + 1}],
                 Line[{x + 1, 𝓎[x + 1]}, 𝕥[]]}, {i, 𝕣}];
 tree = Line[{0, 0}, 𝕥[]];
 symmetrizeRules = Dispatch[Flatten [Function[l,
                    (# -> (# - {0, l[[-1, 2]]/2}))& /@ l] /@
                   Split[Union[DeleteCases[Level[tree, {-2}], {}]],
                         #1[[1]] === #2[[1]]&]]];
 Graphics[tree /. symmetrizeRules /. Line[l__] :> Line[{l}],
          Frame -> True]]
```

9.^{L3} $\varepsilon\varepsilon \to \Sigma\delta\cdots\delta$, Tr$(\gamma_{\mu_1}.\gamma_{\mu_2}.\cdots.\gamma_{\mu_{2n}})$, tanh Identity, Multidimensional Determinant

a) Implement the following identity (meaning the calculation of its right-hand side) for Levi–Civita tensors [269✱] and [41✱] $\varepsilon_{\nu\ldots\pi}$:

$$\varepsilon_{\tau_1\,\tau_2\ldots\,\tau_{r-1}\,\tau_r\,\nu_{r+1}\ldots\,\nu_n}\,\varepsilon_{\tau_1\,\tau_2\ldots\,\tau_{r-1}\,\tau_r\,\mu_{r+1}\ldots\,\mu_n} = r!\,\{\delta_{\nu_{r+1}\,\mu_{r+1}}\,\delta_{\nu_{r+2}\,\mu_{r+2}}\cdots\delta_{\nu_n\,\mu_n}\}_{[\mu_{r+1}\,\cdots\mu_n]}$$

The expression $\{expression\}_{[\mu_{r+1}\,\cdots\mu_n]}$ denotes the Bach bracket and means a complete antisymmetrization in the variables $\mu_{r+1}\ldots\mu_n$.

Here, $\delta_{\nu\mu}$ is the Kronecker symbol, and for all variables with double subscripts, we assume an implicit summation over 1 to *dimension*.

b) Given n matrices ${}^{(k)}A_i^j$ $(i, j, k = 1, \ldots, n)$ of dimensions $n\times n$, the expression [84✱]

$$\{\,{}^{(1)}A_{k_1}^{k_1}\cdots{}^{(n)}A_{k_n}^{k_n}\,\delta_a^b\}_{[k_1,k_2,\ldots,k_n,a]}$$

vanishes identically. Here the expression $\{expression\}_{[\mu_1\,\cdots\mu_n]}$ denotes again the Bach bracket and means a complete antisymmetrization in the variables $\mu_1\ldots\mu_n$, and δ_a^b is the Kronecker symbol. Summation from 1 to n is assumed for all doubly occurring indices. For $n = 2, 3, 4$, verify this identity by explicit calculation. Is $n = 5$ feasible for explicit verification?

c) In many quantum electrodynamics calculations, the trace of the product of Dirac matrices γ_μ, $\mu = 0, 1, 2, 3$ [42✱] has to be calculated. A compact formula for this trace is [318✱], [320✱]

$$\mathrm{Tr}(\gamma_{\mu_1}.\gamma_{\mu_2}.\ldots.\gamma_{\mu_{2n}}) = 4\sum_{\text{all pairings}}\delta_{\text{pairing}}\prod_{\text{all pairs}}\eta_{\mu_i,\mu_j}.$$

Here, the summation extends over all permutations $\{\mu_{i_1}, \mu_{i_2}, \ldots, \mu_{i_{2n}}\}$ of $\{\mu_1, \mu_2, \ldots, \mu_{2n}\}$ such that $\mu_{i_1} < \mu_{i_3} < \cdots < \mu_{i_{2n-1}}$ and $\mu_{i_1} < \mu_{i_2}$, $\mu_{i_3} < \mu_{i_4}$, $\ldots$, $\mu_{i_{2n-1}} < \mu_{i_{2n}}$. The symbol δ_{pairing} is the signature of the permutation $\{\mu_{i_1}, \mu_{i_2}, \ldots, \mu_{i_{2n}}\}$. The inner product extends over all pairs $\{\mu_i, \mu_j\}$. All of the indices μ_i run conventionally from 0 to 3. η_{μ_i,μ_j} is the metric tensor $\eta = \mathrm{diag}\{-1, 1, 1, 1\}$.

Calculate the trace for the product of 2, 4, 6, and 8 Dirac matrices. Use the following representation of the Dirac γ matrices to check the results:

$$\gamma_0 = \begin{pmatrix} 0 & 0 & -i & 0 \\ 0 & 0 & 0 & -i \\ -i & 0 & 0 & 0 \\ 0 & -i & 0 & 0 \end{pmatrix}, \gamma_1 = \begin{pmatrix} 0 & 0 & 0 & -i \\ 0 & 0 & -i & 0 \\ 0 & +i & 0 & 0 \\ +i & 0 & 0 & 0 \end{pmatrix}, \gamma_2 = \begin{pmatrix} 0 & 0 & 0 & -1 \\ 0 & 0 & +1 & 0 \\ 0 & +1 & 0 & 0 \\ -1 & 0 & 0 & 0 \end{pmatrix}, \gamma_3 = \begin{pmatrix} 0 & 0 & -i & 0 \\ 0 & 0 & 0 & +i \\ +i & 0 & 0 & 0 \\ 0 & -i & 0 & 0 \end{pmatrix}.$$

d) The following identity holds for almost all complex z_k [283★]:

$$\prod_{\substack{k,l=1 \\ k<l}}^{n} \tanh(z_k - z_l) = \frac{1}{2^{\lfloor n/2 \rfloor} \lfloor n/2 \rfloor !} \sum_{\sigma} \text{signature}(\sigma) \prod_{k=1}^{\lfloor n/2 \rfloor} \tanh(z_{\sigma(2k-1)} - z_{\sigma(2k)})$$

The summation runs over all elements of the permutations σ of $\{1, 2, \ldots, n\}$. Prove this identity for $n = 6$. (Do not use functions like `Simplify`, `Together`, `TrigToExp`, etc., but only functions discussed so far.)

e) The determinant of a (2D) $n \times n$ matrix $\mathbb{A}$ with elements $a_{i,j}$ can be written in the form

$$\det(\mathbb{A}) = \varepsilon_{1,2,\ldots,n}\, \varepsilon_{k_1,k_2,\ldots,k_n}\, a_{1,k_1}\, a_{2,k_2} \cdots a_{n,k_n}$$

where summation from 1 to n is understood for the doubly occurring variables $k_1, \ldots, k_n$ and $\varepsilon_{1,2,\ldots,n}$ is fully antisymmetric in all of its variable pairs. This suggests the generalization of the determinant for an d-dimensional (dD) $n \times n \ldots \times n$ "matrix" $\mathbb{A}_d$ with elements $a_{i_1,i_2,\ldots,i_d}$ of the form [155★], [231★], [163★], [236★], [130★], [295★], [207★]

$$\det(\mathbb{A}_d) = \prod_{j=1}^{d} \varepsilon_{k_1^{(j)},k_2^{(j)},\ldots,k_n^{(j)}} \prod_{i=1}^{n} a_{k_i^{(1)},k_i^{(2)},\ldots,k_i^{(m)}}$$

where $k_m^{(1)} = m$ and again summation from 1 to n is understood for the doubly occurring variables. Implement a function `MultiDimensionalDet` that for a given dD matrix $\mathbb{A}_d$ of size $n \times n \ldots \times n$ calculates this multidimensional determinant.

10.^{L1} Digits in π, Mediant Insertion, Matrix Product

a) Let $l_{ij}(\pi)$ be the sequence of number pairs $\{\{i_1, j_1\}, \{i_2, j_2\}, \ldots\}$ of the positions of the first appearance of the digit i after the digit j in the decimal expansion of π [239★], where $i_1 < j_1 < i_2 < j_2 < \ldots$. program the computation of the $l_{ij}(\pi)$.

b) Given a list l of (reduced) rational numbers, write a function that inserts the mediant between each two numbers of the list l. The mediant of two rational numbers p_1/q_1 and p_2/q_2 (where $\gcd(p_1, q_1) = \gcd(p_2, q_2) = 1$) is defined as the number $(p_1 + p_2)/(q_1 + q_2)$.

c) The constant e can be calculated through the following limit of matrix product [83★], [129★]

$$\begin{pmatrix} a_n & c_n \\ b_n & d_n \end{pmatrix} = \prod_{k=1}^{n} \begin{pmatrix} 2k & 2k-1 \\ 2k-1 & 2k-2 \end{pmatrix}$$

$$e = \lim_{n \to \infty} \frac{a_n + c_n}{b_n + d_n}.$$

(This is basically an unfolded continued fraction expansion). How large a n is needed to obtain 1000 correct digits of e?

11.^{L1} d'Hondt Voting

Implement the d'Hondt's voting method. If possible, try not to use any temporary variables. The d'Hondt's voting method is as follows: Suppose a parliament with a given number of seats is to be filled with representatives from several parties on the basis of an election. Divide the number of votes received by each party by 1, 2, 3, etc. Put the resulting numbers in decreasing order (where each number is included according to its multiplicity). Now, assign one seat to the party with the largest number, one seat to the party with the second largest number, etc., until all seats have been assigned. If, at the end, more equal numbers than seats are available, choose the parties to get the seats randomly. (We discuss the generation of random numbers in detail in the next chapter, so either do not treat this possibility at the moment or come back to this later.)

Here is an example. Suppose six seats are to be assigned and that the results of the election are: A received 8 votes, B received 5, and C received 9 votes. We get the following table of numbers after dividing by 1, 2, 3 …:

$$\begin{array}{lllllll} A: & 8 & 4 & \frac{8}{3} & \frac{8}{4} & \frac{8}{5} & \frac{4}{3} & \cdots \\ B: & 5 & \frac{5}{2} & \frac{5}{3} & \frac{5}{4} & \frac{5}{5} & \frac{5}{6} & \cdots \\ C: & 9 & \frac{9}{2} & \frac{9}{3} & \frac{9}{4} & \frac{9}{5} & \frac{3}{2} & \cdots \end{array}$$

Then, the decreasing list (with corresponding parties) is

$$\begin{array}{cccccc|c} 9 & 8 & 5 & \frac{9}{2} & 4 & 3 & \frac{8}{3} \\ C & A & B & C & A & C & A \end{array}$$

Thus, A gets two seats, C gets three, and B gets one seat. For more on the interesting mathematical aspects of elections, see [261✱], [35✱], [297✱], [260✱], [27✱], [97✱], [329✱], [262✱], [287✱], [159✱], [99✱], [169✱], and [296✱]; for an interesting nonpolitical application, see [309✱]. For a nice *Mathematica*-based proof of the related Arrow's theorem, see [293✱].

12.^{L2} Grouping, Unsorted Complements

a) Suppose we want to divide a given a list of real (complex) numbers into groups of numbers that are "close together". Program a corresponding function.

b) Given a list of real positive integers $\{z_1, z_2, \ldots, z_n\}$ and a positive number ϵ, extract all possible maximal length chains of numbers $\{z_{i_1}, z_{i_2}, \ldots, z_{i_j}\}$, $j \geq 2$, such that $|z_{i_{k+1}} - z_{i_k}| \leq \epsilon$. Do not make use of the built-in function `Split`.

c) Given a list of lists with vector-valued elements. (The vectors are lists (of equal length) of real numbers.) Assume some vectors occur possibly more than once, but the components of the vectors are slightly different (`Equal` would return `True`, but the last digits might be different). Write an efficient `VectorUnion` function that unions the list of vectors. Why is it possible to implement a function more efficient than the built-in `Union`?

d) Write a function `UnSortedComplement[`l_1`, `l_2`]`, that forms the complement of l_1 with respect to the list l_2 and does not reorder the list l_1 and takes the multiplicity in the list l_2 into account when removing elements from l_1.

13.^{L1} All Arithmetic Expressions

Given a list of numbers (atoms) and a list of binary operations, use the numbers and the operations to form all syntactically correct nested expressions. The order of the numbers should not be changed, and only the binary operations and parentheses `()` should be inserted between the numbers [72✱].

14.^{L1} Symbols with Values, `SetDelayed` Assignments, Counting Integers

a) Identify which values will be collected in the following list `li`:

```
name = DeleteCases[Names["*"], "names"];

li = {};

Do[Clear[f];
   f[Evaluate[ToExpression[names[[i]] <> "_"]]] =
                                          ToExpression[names[[i]]]^2;
   If[f[3] =!= 9, AppendTo[li, {names[[i]], ToExpression[names[[i]]]}]],
  {i, 1, Length[names]}];

li
```

b) Identify which built-in functions will be returned from the following code:

```
Cases[{#, ToExpression[StringJoin["f[x_] :=" <> # <> "[x]"]];
          StringPosition[ToString[FullForm[DownValues[f]]], #]}& /@
                                        Names["System`*"], {_, {}}]
```

c) Given the following list of numbers

```
Do[data[n] = Table[IntegerPart[k Sin[k]], {k, 10^n}], {n, 4}].
```

Use various implementations to count how often each integer appears in `data[`*n*`]`.

15.^{L1} `Sort[`***list***`,` ***strangeFunction***`]`

Examine whether `Sort` generates error messages for nontransitive, asymmetric order relations.

16.^{L3} Bracket-Aligned Formatting, Fortran Real*8, `Method` Option, Level functions, Conversion to `StandardForm` inputs

a) Write a function that formats a *Mathematica* expression in such a way that pairs of corresponding brackets `[` and `]` of the `FullForm` are aligned.

b) *Mathematica* includes the command `FortranForm`. Unfortunately, no type declarations are allowed, so the output is not always in an appropriate form to be given directly to a Fortran program. Write a function that rewrites arbitrary integers in the form of Fortran Real*8 numbers. Let the result be a string.

c) Which *Mathematica* functions have a `Method` option? Can one use the *Mathematica* kernel to find the possible option settings?

d) Which *Mathematica* functions take level specifications? Can one use the *Mathematica* kernel to find these functions?

e) Write a function that converts the `InputForm` input cells of the *GuideBooks* into `StandardForm` cells. Preserve all comments, indentation (as much as possible) and use typical `StandardForm` symbols such as π, ⅈ, ⅇ, and →.

17.^{L2} `ReplaceAll` Order, Pattern Realization, Pure Functions

a) The function `orderedTriedExpressions` returns a list of all (sub)expressions of *expr* in the order tried by `ReplaceAll`.

```
orderedTriedExpressions[expr_] :=
 Module[{bag = {}}, expr /. x_ :> Null /; (AppendTo[bag, x]; False); bag]
```

Implement a version of `orderedTriedExpressions` that uses only built-in functions. Implement another version of `orderedTriedExpressions` that does not make any assignments (no `Set` or `SetDelayed`).

b) Write a function `patternRealization` that, analogous to `MatchQ`, takes two arguments *expression* and *expressionWithPatterns* and gives a list of the actual realizations of the pattern variables. Write a version of `pattern‐Realization` that does not contain any auxiliary variables. Test both versions on a few examples.

c) Given an expression that contains one-argument pure functions using `Slots` (like `#^2&[(#1 + #2)^3&[#1, 2#1]&[(#1 + #2 + (#^2&[#]))&[#1, #4]]]&`), write a function that replaces these pure functions with ones that have two arguments, and use explicit variables (like `Function[`*x*`, ` *bodyContainingx*`]`).

18.^{L3} Matrix Identities, Frobenius Formula, Iterative Matrix Square Root

a) For an arbitrary 3×3 matrix $\mathbf{A}$,

$$\mathbf{A}^3 - \operatorname{tr}(\mathbf{A})\,\mathbf{A}^2 + \frac{1}{2}\left(\operatorname{tr}(\mathbf{A})^2 - \operatorname{tr}\left(\mathbf{A}^2\right)\right)\mathbf{A} - \det(\mathbf{A})\,\mathbf{1} = 0,$$

where tr is the trace, det is the determinant, and $\mathbf{1}$ is the 3D identity matrix. (This identity follows from the Theorem of Cayley–Hamilton together with the Newton relations.) Prove this identity.

b) For arbitrary 2×2 matrices $\mathbf{A}$ and $\mathbf{B}$, the following identity hold [20✱]:

$$\boldsymbol{B}.\boldsymbol{A} = (\operatorname{tr}(\boldsymbol{A}.\boldsymbol{B}) - \operatorname{tr}(\mathbf{A})\operatorname{tr}(\mathbf{B}))\,\mathbf{1} + \operatorname{tr}(\mathbf{A})\,\mathbf{B} + \operatorname{tr}(\mathbf{B})\,\mathbf{A} - \boldsymbol{A}.\boldsymbol{B}$$

Here again tr stands for the trace, and $\mathbf{1}$ is the 2D identity matrix. Prove this identity. Does it also hold for 3×3 matrices? If not, does there exist a generalization of the form

$$\boldsymbol{B}.\boldsymbol{A} = \left(\sum_{i,j,k,l=0}^{1} c_{i,j,k,l}^{(\mathbf{1})} \operatorname{tr}\left(\mathbf{A}^i.\mathbf{B}^j\right)\operatorname{tr}(\mathbf{A})^k \operatorname{tr}(\mathbf{B})^l\,\mathbf{1}\right) + \left(\sum_{\alpha=1}^{2}\sum_{i,j,k,l=0}^{1} c_{i,j,k,l}^{(\boldsymbol{A},\alpha)} \operatorname{tr}\left(\mathbf{A}^i.\mathbf{B}^j\right)\operatorname{tr}(\mathbf{A})^k \operatorname{tr}(\mathbf{B})^l\,\mathbf{A}^\alpha\right) +$$
$$\left(\sum_{\alpha=1}^{2}\sum_{i,j,k,l=0}^{1} c_{i,j,k,l}^{(\boldsymbol{B},\alpha)} \operatorname{tr}\left(\mathbf{A}^i.\mathbf{B}^j\right)\operatorname{tr}(\mathbf{A})^k \operatorname{tr}(\mathbf{B})^l\,\mathbf{B}^\alpha\right) - \boldsymbol{A}.\boldsymbol{B}$$

for 3×3 matrices?

c) Prove the Amitsur–Levitzky identity [38✱] for $n = 3$. The Amitsur–Levitsky identity states that for the $2\,n$ matrices of dimension $n\times n$, denoted by $\mathbf{A}_1, \mathbf{A}_2, \ldots, \mathbf{A}_{2n}$, the following sum over all permutations σ of the numbers $(1, 2, \ldots, 2\,n)$ yields the $n\times n$ zero matrix $\mathbf{0}_n$: $\sum_\sigma \operatorname{signature}(\sigma)\,\mathbf{A}_{\sigma(1)}.\mathbf{A}_{\sigma(2)}.\cdots.\mathbf{A}_{\sigma(2n)} = \mathbf{0}_n$.

d) Fix a univariate polynomial $q(x)$ of degree n and consider the eigenvalue problem [204✱]

$$\frac{\partial^k\left(q(x)\,\psi_j^{(n,k)}(x)\right)}{\partial x^k} = \lambda_j^{(n,k)}\,\psi_j^{(n,k)}(x).$$

Assume that the $\psi_j^{(n,k)}(x)$ are polynomials too and conjecture a closed form for the eigenvalues $\lambda_j^{(n,k)}$.

e) The well-known Frobenius formula [98✱], [114✱] expresses the inverse of a 2×2 block matrix $\begin{pmatrix} \mathbb{A} & \mathbb{B} \\ \mathbb{C} & \mathbb{D} \end{pmatrix}$ ($\mathbb{A}$, $\mathbb{B}$, $\mathbb{C}$, and $\mathbb{D}$ being $n\times n$ matrices) in the form

$$\begin{pmatrix} \mathbb{A}^{-1} - \mathbb{A}^{-1}.\mathbb{B}.\left(\mathbb{B} - \mathbb{A}.\mathbb{C}^{-1}.\mathbb{D}\right)^{-1} & -\mathbb{A}^{-1}.\mathbb{B}.\left(\mathbb{D} - \mathbb{C}.\mathbb{A}^{-1}.\mathbb{B}\right)^{-1} \\ \left(\mathbb{B} - \mathbb{A}.\mathbb{C}^{-1}.\mathbb{D}\right)^{-1} & \left(\mathbb{D} - \mathbb{C}.\mathbb{A}^{-1}.\mathbb{B}\right)^{-1} \end{pmatrix}.$$

(The last expression can be rewritten in various equivalent forms.) Here we assume that the inverses of all four block matrices $\mathbb{A}$, $\mathbb{B}$, $\mathbb{C}$, and $\mathbb{D}$ exist. Implement a function that derives this type of representation for an $n\times n$ block matrix. Test the function for $n = 2$, 3, and 4.

f) Let $\mathbf{A}$ be an $n\times n$ matrix. Its square root can be calculated by iterating the map $\mathbf{B} \to \mathbf{B}.(\mathbf{B}.\mathbf{B} + 3\,\mathbf{A})\,(3\,\mathbf{B}.\mathbf{B} + \mathbf{A})^{-1}$ starting with an nD identity matrix [180✱]. Use this iteration to calculate a numerical approximation to the square root of a 10×10 Hilbert matrix.

g) Consider the following three $n\times n$ matrices $\mathbf{G}(a, b)$, $\mathbf{W}(x)$, and $\mathbf{M}(x_1, \ldots, x_n)$ with elements

$$g_{i,j}(a, b) = \int_a^b f_i(x)\, f_j(x)\, dx$$

$$w_{i,j}(x) = \frac{\partial^{i-1} f_j(x)}{\partial x_j^{i-1}}$$

$$m_{i,j}(x_1, \ldots, x_n) = f_j(x_j).$$

Here the f_j are unspecified real-valued functions. Verify the following relations for small n by explicit calculation [63✱]:

$$\frac{\partial^{n^2} \det(\mathbf{G}(a, b))}{\partial b^{n^2}}\Big|_{b=a} = \frac{\prod_{k=1}^{2n-1} k^{n-|n-k|}}{n^2\,!} \det(\mathbf{W}(a))$$

$$\det(\mathbf{G}(a, b)) = \int_a^b \cdots \int_a^b \det(\mathbf{M}(x_1, \ldots, x_n))^2\, dx_1 \ldots dx_n$$

$$\det(\mathbf{W}(x)) = \frac{\partial^{n(n-1)/2} \det(\mathbf{M}(x_1, \ldots, x_n))}{\partial x_2\, \partial x_3^2 \cdots \partial x_n^{n-1}}$$

where $1 \leq i,\, j \leq n$. How far can one go with n?

h) Define the derivative $\mathbb{d}$ of a function f of a matrix $\mathbf{A}$ through the component representation [69✱], [70✱], [57✱], [272✱], [149✱]

$$\left(\frac{\mathbb{d} f(\mathbf{A})}{\mathbb{d}\mathbf{A}}\right)_{ijkl} = \frac{d\,(f(\mathbf{A}))_{ij}}{d\,(\mathbf{A})_{kl}}.$$

(This means this matrix derivative is a tensor of depth 4.) Implement this derivative in an index-free manner.

For the power function $f(z) = z^n$, the derivative can be expressed as

$$\left(\frac{\mathbb{d}\mathbf{A}^n}{\mathbb{d}\mathbf{A}}\right)_{ijkl} = \sum_{m=1}^{n} \left(\mathbf{A}^{m-1}\right)_{ik} \left(\mathbf{A}^{n-m}\right)_{lj}.$$

Implement this formula also in an index-free manner.

19.^{L2} Autoloading and Package Test

a) Many *Mathematica* functions are programmed in the *Mathematica* language and autoloaded when needed. Find these functions.

b) Analyze all packages from the standard packages directory according to the following criteria:
- How many commands are exported?
- Are undocumented commands exported?
- How many variables are used inside the packages?
- Which packages export no commands?
- Which packages change the attributes of built-in commands?
- Which packages alter the options for the built-in commands?
- Which packages give error messages when loaded?

Do not load each individual package “manually”.

20.^{L2} `PrecedenceForm`

Examine the meaning of the built-in command `PrecedenceForm`, and determine the precedence of all built-in commands (when possible). Knowing preferences is important, for instance, for determining if `2 + 4 // #; &` means `2 + 4 // (#; )&` or `(2 + 4 // #); &` and so on. Do the same with all named characters (like `CircleTimes`, `DoubleLongRightArrow`) that can be used as operators.

21.^{L2} One-Liners

a) Write a “one-liner” that makes the following: Given a positive integer sum s and a list *summands* of positive integers a_i, the function `AllPossibleFactors[`s`, `*summands*`]` should return a list of all possibilities of lists of factors $\{f_1, f_2, \ldots, f_n\}$, such as

$$s \leq \sum_{i=0}^{n} f_i\, a_i, \text{ with } f_i \geq 0 \quad \forall\, i$$

(A one-liner is, “by definition”, a *Mathematica* program that consists only of one piece of code, uses no named or temporary variables or functions, and is often a nice example in functional programming. A one-liner does not necessarily fit into one line.) How many different arrangements of 1¢, 5¢, 10¢, and 25¢ coins can one make, so that the net total is less than $1?

b) Write a one-liner for the Ferrer conjugate from Exercise 9.d) in Chapter 5.

c) Model the function `AppendTo` by using the function `Append`.

d) Write a one-liner that recursively implements Meissel’s formula [331✱], [223✱], [43✱], [208✱], [209✱] for the calculation of $\pi(n)$, the number of primes less than or equal to n.

$$\pi(n) = n - 1 + \pi\left(\sqrt{n}\right) + \sum_{j=1}^{k} (-1)^j \sum_{p_{i_1} < p_{i_2} \cdots < p_{i_j}} \left\lfloor \frac{n}{p_{i_1} \cdots p_{i_j}} \right\rfloor.$$

Here $p_1, p_2, \ldots, p_k$ are all primes less than or equal to $\sqrt{n}$. (The nth prime can be obtained using `Prime[`n`]`.) Use only built-in symbols.

e) Write a one-liner for generating the following polynomials $p_n(x_1, \ldots, x_n)$ [184✱] in factored form:

$$p_n(x_1, \ldots, x_n) = \sum_\sigma \prod_{k=1}^{n} x_k^{\mu_k(\sigma)}$$

The summation runs over all elements of the permutations σ of $\{1, 2, \ldots, n\}$. For a permutation $\sigma = \{j_1, j_2, \ldots, j_n\}$ the function $\mu_k(\sigma)$ counts the number of j_l, $l = k+1, \ldots, n$ that are smaller than j_k. Calculate p_1 to p_8 explicitly.

f) Write a one-liner that, for given positive integers k and p ($0 < p < k$) proves the following identity [39✱]:

$$\sum_{n_1=1}^{k-1} \sum_{n_2=1}^{k-n_1-1} \sum_{n_3=1}^{k-n_1-n_2-1} \cdots \sum_{n_p=1}^{k-n_1-n_2-\ldots-n_p-1} \frac{1}{n_1!} \frac{\partial f(x)^{n_1}}{\partial x^{n_1}} \frac{1}{n_2!} \frac{\partial f(x)^{n_2}}{\partial x^{n_2}} \cdots \frac{1}{n_p!} \frac{\partial f(x)^{n_p}}{\partial x^{n_p}}$$

$$\frac{1}{(k-n_1-n_2-\cdots-n_p)!} \frac{\partial f(x)^{k-n_1-n_2-\cdots-n_p}}{\partial x^{k-n_1-n_2-\cdots-n_p-1}} = (p+1) \frac{(k-1)!}{(k-1-p)!} \frac{1}{k!} \frac{\partial^{k-p-1} f(x)^k}{\partial x^{k-p-1}}.$$

g) For any $n \times n$ matrix $\mathbb{A}$, the following identity holds [8✱]:

$$\det(\mathbb{A}) = \frac{1}{n!} \det \begin{pmatrix} \operatorname{tr}(\mathbb{A}) & 1 & 0 & \cdots & 0 & 0 \\ \operatorname{tr}(\mathbb{A}^2) & \operatorname{tr}(\mathbb{A}) & 2 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ \operatorname{tr}(\mathbb{A}^{n-1}) & \operatorname{tr}(\mathbb{A}^{n-2}) & \operatorname{tr}(\mathbb{A}^{n-2}) & \cdots & \operatorname{tr}(\mathbb{A}) & n-1 \\ \operatorname{tr}(\mathbb{A}^n) & \operatorname{tr}(\mathbb{A}^{n-1}) & \operatorname{tr}(\mathbb{A}^{n-2}) & \cdots & \operatorname{tr}(\mathbb{A}^2) & \operatorname{tr}(\mathbb{A}) \end{pmatrix}.$$

Implement a one-liner that checks this identity for a given matrix $\mathbb{A}$. ($n!$ is the factorial function, in *Mathematica* $n!$.)

h) Let $p_{\mathbb{A}}(z) = \sum_{j=0}^{n} c_j z^j$ be the characteristic polynomial of the $n \times n$ matrix $\mathbb{A}$. Then, the inverse $\mathbb{A}^{-1}$ can be expressed as $\mathbb{A}^{-1} = -c_0^{-1} \sum_{j=1}^{n} c_j \mathbb{A}^{j-1}$ [8✱]. Implement a one-liner that uses this identity to calculate the inverse.

i) Write a one-liner that implements the expansion of an arbitrary function $f(z)$ in the product [18✱]

$$\Pi_o(f(z), z_0) = \prod_{k=0}^{o} (\mathcal{E}_k(f(z_0)))^{\frac{\ln(z/z_0)}{k!}}$$

around the point $f(z_0) \neq 0$ and where $\mathcal{E}_0(f(\zeta)) = f(\zeta)$ and $\mathcal{E}_k(f(\zeta)) = \exp(\zeta\, \partial \ln(f(\zeta)) / \partial \zeta)$. For a sufficiently smooth function $f(z)$, we have $\lim_{o \to \infty} \Pi_o(f(z), z_0) = f(z)$. Calculate $\Pi_{12}(\cos(\pi/2), 1)$.

j) Write a one-liner that generates all different expressions resulting from the repeated application of a binary (two-argument) function to n sorted arguments. For example, for the four arguments a, b, c, d and the function f, the five compositions $f(a, f(b, f(c, d)))$, $f(a, f(f(b, c), d))$, $f(f(a, b), f(c, d))$, $f(f(a, f(b, c)), d)$, and $f(f(f(a, b), c), d)$ should be formed. How frequently do k consecutive closing ')' occur for ten arguments? For six equal arguments $a = b = \ldots = \sqrt{-1}$, and f=`Power`, how many numerically different expressions result [104✱], [122✱]?

k) Write a one-liner `KolakoskiSequence[n]` that calculates the first n terms of the Kolakoski sequence [167✱], [68✱], [279✱]. With the exception of n, the function `KolakoskiSequence` should not use any not built-in commands. The Kolakoski sequence {2, 2, 1, 1, 2, 1, 2, 2, 1, 2, 2, 1, 1, …} is the (unique) sequence of its own run lengths (meaning 2 twos, then 2 ones, then 1 two, 1 one, then 2 twos, ….

l) Given the three differential identities

$$x'(t) = y(t)\, z(t)$$
$$y'(t) = x(t)\, z(t)$$
$$z'(t) = x(t)\, y(t)$$

define a sequence of functions recursively through $\sigma_k(t) = \partial\sigma_{k-1}(t)/\partial t$, starting with $\sigma_0(t) = x(t)$. The resulting $\sigma_n(t)$ have the form $\sigma_n(t) = \sum_{i,j,k=0}^{n+1} c_{i,j,k}\, x(t)^i\, y(t)^j\, z(t)^k$. The coefficients fulfill the following sum rule: $\sum_{i,j,k=0}^{n+1} c_{i,j,k} = n!$ [79✱]. Implement a one-liner `factorialSumTest` that, by explicit calculation, checks this property for a given n (the factorial of n is just `n!` in *Mathematica*). The implementation should not use any built-in symbol. Check the sum property for $0 \le n \le 100$.

m) Write a one-liner that, for a given n, calculates the number of permutations having k ($k = 0, 1, \ldots, n$) increasing two-sequences in all permutations of $\{1, 2, \ldots, n\}$. (An increasing two sequence in a permutation $\{j_1, j_2, \ldots, j_n\}$ is a pair $\{j_i, j_{i+1}\}$ such that $j_{i+1} = j_i + 1$ [150✱], [151✱].)

22.^{L2} Precedences

a) What are the results of the following expressions?

```
Function[x, Hold[x], {Listable}] @
            Hold[{1 + 1, 2 + 2, 3 + 3}]

Function[x, Hold[x], {Listable}] @@
             Hold[{1 + 1, 2 + 2, 3 + 3}]

Function[x, Hold[x], {Listable, HoldAll}] @
             Hold[{1 + 1, 2 + 2, 3 + 3}]

Function[x, Hold[x], {Listable, HoldAll}] @@
             Hold[{1 + 1, 2 + 2, 3 + 3}]

Function[x, Hold[x], {Listable, HoldAll}] @
            (#& @@ Hold[{1 + 1, 2 + 2, 3 + 3}])

Function[x, Hold[x], {Listable}] @@
             (#& @@ Hold[{1 + 1, 2 + 2, 3 + 3}])

Function[x, Hold[x], {Listable, HoldAll}] @ #& @@
             Hold[{1 + 1, 2 + 2, 3 + 3}]

Function[x, Hold[x], {Listable, HoldAll}] @
    Function[x, Hold[x], {Listable, HoldAll}] @@
                 Hold[{1 + 1, 2 + 2, 3 + 3}]

Function[x, Hold[x], {Listable, HoldAll}] @@
    Function[x, Hold[x], {Listable, HoldAll}] @
                 Hold[{1 + 1, 2 + 2, 3 + 3}]

Function[x, Hold[x], {Listable, HoldAll}] @@
    Function[x, Hold[x], {Listable, HoldAll}] @@
                 Hold[{1 + 1, 2 + 2, 3 + 3}]
```

b) If

```
localVar = 11;
```

```
Block[{localVar = 1}, Print[localVar]; WhatIsHere]
```

prints out 11, what might have been coded in *WhatIsHere*? Find a *WhatIsHere* that also works if `Block` is replaced by `With`.

23.^{L2} Puzzles

a) What is the result of the following input? (Here the spaces in the input matter; do not introduce or remove blanks.)

```
1 @ 2 @@ 3 / 4 /@ 6 //@ 7 || 8 | 9 /.10 /.11
```

b) Find a value for `factor`, such that the following two definitions for give different results.

```
scaledReversedShiftedListV1[factor_, list_List] :=
    Function[Join[factor #, Reverse[factor/2 #]]][list]

scaledReversedShiftedListV2[factor_, list_List] :=
    Function[x, Join[factor x, Reverse[factor/2 x]]][list]
```

c) Predict the result of the following input.

```
{#, InputForm[ToExpression @ #],
    FullForm[ToExpression @ #]}& /@
              Table["1"<>Table[".", {i}], {i, 1, 11}] // TableForm
```

d) Predict the result of the following input.

```
Power @@ Unevaluated[Times[2, 2, 2]].
```

e) Predict the result of the following input.

```
Power[Delete @@ Cos[Sin[2], 0]].
```

f) Predict the result of the following input.

```
{Dimensions[#], Length[Flatten[#]]}& /@
                  NestList[Outer[List, #, #]&,{1., 2}, 3]
```

g) Given a held expression, write a function that replaces all occurrences of *p*`_Plus` by the evaluated result of `Length[`*p*`]`.

h) Predict the result of the following input.

```
  Block[{Infinity}, Apply[Subtract, {Infinity, Infinity}]]
```

i) Predict the result of the following input.

```
inherit[fNew_, fOld_] :=
CompoundExpression[
 SetAttributes[fNew, Attributes[fOld]];
 Options[fNew] = Options[fOld];
 (#[fNew] = (#[fOld] /. fOld -> fNew))& /@
 {NValues, SubValues, DownValues,
  OwnValues, UpValues, FormatValues}]

SetAttributes[f, {Listable}];
f[x_Plus] := Length[Unevaluated[x]];
```

```
Module[{f},
       inherit[f, ToExpression["f"]];
       SetAttributes[f, HoldAll];
       f[i__Integer] = i^2;
       f @@ f[{1 + 1, 2 + 2}]]
```

j) Predict which messages will be issued when evaluating the following:

```
Evaluate //@ Block[{I = 1}, I^2]
```

What will be the result?

k) Find a *Mathematica* expression *expr* such that `First[`*expr*`]` and *expr*`[[1]]` give different results.

l) Predict the result of the following inputs:

```
f[x_] := Block[{α = Not[TrueQ[α]]}, f[x + 1] /; α]
f @@ f[0]
```

m) Predict the result of the following input:

```
Module[{x = D, f}, C @@ f[x_] ~ Set ~ x // f[C]&] -
Module[{x = D, f}, Set @@ f[x_] ~ C ~ x // f[C]&]
```

n) Predict the result of the following input:

```
ℂ = 0;
Union[Array[1&, {100}], SameTest -> ((ℂ = ℂ + 1; False)&)];
ℂ
```

o) Implement a function `virtualMatrix[`*dim*`]` that generates a "virtual" matrix of size $dim \times dim$ that behaves like a "real" matrix as in the following:

```
In[2]:= M = virtualMatrix[10^6];

In[3]:= {MatrixQ[M], Dimensions[M], Length[M[[1]]],
          {M[[1, 1]], M[[-1, -1]]},
          M[[1000, 1000]] = 1000; M[[1000, 1000]]}

Out[3]= {True,{1000000,1000000},1000000,{1,1},1000}
```

Do not unprotect any built-in function or use upvalues.

p) Given an expression *expr* (fully evaluated and not containing any held parts) and two integers k and l, what is the result of `MapIndexed[(Part[`*expr*`, ##]& @@ #2)&, `*expr*`, {`*k*`, `*l*`}, Heads -> True]`?

24.^{L2} Hash Value Collisions, Permutation Digit Sets

a) The function `Hash[`*expr*`]` returns the hash value of *expr*. Find two integers that are hashed to the same hash value.

b) Let $\mathcal{S}_o^{(b)}$ be the set of all o-digit integers in base b with every digit from the range $[1, o]$ appearing exactly once in the base b representation. (For instance $\mathcal{S}_3^{(10)} = \{123, 132, 213, 231, 312, 321\}$.) Let $\mathcal{M}_o^{(b)}$ be the set of pairs $\{s_1, s_2\}$, $s_1, s_2 \in \mathcal{S}_o^{(b)}$ such that $s_2 = m\, s_1,\ j \in \mathbb{N},\ m \geq 2$. Find the sets $\mathcal{M}_o^{(b)}$ for $2 \leq b \leq 10$, $1 \leq k \leq b-1$. How many elements does $\mathcal{M}_{11}^{(12)}$ have? Format the results in the form $s_2 = m\, s_1$.

25.^{L1} Function Calls in GluedPolygons

In the construction of the glued polygons in Section 6.0, the function Trace was used to show the relative frequency of the use of various list-manipulating functions. This was overestimating the number of function calls. Determine the actual number of calls to the functions Reverse, Join, Dot, Map, Partition, Apply, Take, MapThread, Drop, Table, Part, and Flatten when evaluating GluedPolygons[5, 3Pi/4, 1, Polygon, Display⋱
Function -> Identity].

Solutions

1. Benford's Rule

We cannot give a completely general solution here. First, we have to read in the data using Get or ReadList depending on the nature of the data. Then, we must extract the first digits. Depending on the kind of data, we may use First[⋅
IntegerDigits[...]], First[RealDigits[...]], or First[ToExpression[...]]. These have to be applied to the list containing the data using Map, and then Cases[..., *digit*] and/or Count[..., *digit*] finds the number of digits. If the reader does not have any data, the reader can look in the package Miscellaneous`Chemi⋱
calElements`. (A representative collection of data can also be found in [14✱], but it has to be typed in; also, some web hosts have some data relevant to this exercise, like the ionization energies of atoms http://www.physik.uni-kassel.de/theorie/plasma/.) First, we load the package and then define three auxiliary functions data, firstNumber, and counter. Their use is obvious.

```
Needs["Miscellaneous`ChemicalElements`"]

(* extract data for all elements *)
data[what_] := (what /@ Elements)

(* handling integers and real numbers differently *)
firstNumber[number_] :=
Which[MachineNumberQ[number], RealDigits[number][[1, 1]],
      IntegerQ[number], IntegerDigits[number][[1]],
      (* ignore data *) True, Sequence @@ {}]

(* count first digits *)
counter[dat_] := {#, Count[dat, #]}& /@ {1, 2, 3, 4, 5, 6, 7, 8, 9}
```

We now analyze the atomic weight, melting point, boiling point, heat of fusion, heat of vaporization, density, and thermal conductivity for the frequency of appearance of the digits 1 through 9 in the first place.

We turn off some of the warning and error messages from this package because they appear so frequently. The result of the following counts are lists with elements of the form {*firstDigit*, *numberOfItsAppearance*}.

```
Off[AtomicWeight::unstable]; Off[AtomicWeight::unknown];
aw = counter[firstNumber /@ data[AtomicWeight]]

(* counter making function *)
makeCounter[property_] :=
counter[firstNumber /@ (If[# =!= Unknown, #[[1]],
                             Sequence @@ {}]& /@ data[property])]
```

```
Off[MeltingPoint::form]; Off[MeltingPoint::unknown];
mp = makeCounter[MeltingPoint]

Off[BoilingPoint::form]; Off[BoilingPoint::unknown];
bp = makeCounter[BoilingPoint]

Off[HeatOfFusion::form]; Off[HeatOfFusion::unknown];
hf = makeCounter[HeatOfFusion]

Off[HeatOfVaporization::form]; Off[HeatOfVaporization::unknown];
hv = makeCounter[HeatOfVaporization]

Off[ThermalConductivity::form]; Off[ThermalConductivity::unknown];
tc = makeCounter[ThermalConductivity]

Off[MessageName[Density, #]]& /@ {"form", "temp", "tempform", "unknown"};
d = makeCounter[Density]
```

Here are all results combined. The *i*th element of the following list is the number of occurrences of the digit *i*.

```
Plus @@ (Transpose[#][[2]]& /@ {aw, mp, bp, hf, hv, tc, d})
```

Here is a comparison of the calculated frequency with the theoretical prediction of the relative frequencies.

```
% / Plus @@ % // N

Table[Log[10, 1 + 1/n], {n, 1, 9}] // N
```

Note that the theoretical probabilities, of course, add up to 1.

```
N[Plus @@ %]
```

In view of the small set of data, the degree of agreement is astounding. Benford’s rule is often correct even for numbers generated purely mathematically.

We now read it population data of US cities and villages from 2003 and earlier. The following web page contains links to files with the data for all states.

```
topPage =
Import["http://www.census.gov/popest/cities/SUB-EST2003-04.html", "Text"];
```

There are the link names of the files with the population data.

```
tableURLs = StringJoin["http://www.census.gov/popest/cities/", #]& /@
    StringReverse /@ StringCases[StringReverse @ topPage,
        ShortestMatch[StringReverse["csv"] ~~ __ ~~
                      StringReverse["tables/SUB-EST2003"]]];
```

Here are a few of the data shown.

```
StringTake[Import[tableURLs[[38]],"Text"], {13005, 13339}]
```

We import the population data of more than 8000 cities and villages and count how often the first digit is the integer *k*.

```
allData =
Table[(* read in file as a string *)
      dataSet = Import[tableURLs[[k]], "Text"];
      (* extract population data *)
      townVillageData = StringReplace[#, "," -> ""]& /@
         (* treat strings and towns differently *)
          (StringReplace[#, If[StringCount[#, "\""] === 0,
                              "," -> ", ", "\"" -> " "]]& /@
          StringCases[dataSet,
                      ShortestMatch[("town" | "village") ~~ __ ~~ "\n"]]);
      (* use latest available population data *)
      If[# =!= {}, Last[#], {}]&[
        StringCases[StringSplit[#], NumberString]]& /@ townVillageData,
        {k, Length[tableURLs]}];

(* count occurrences of first digits 1 to 9 *)
{First[#], Length[#]}& /@
  Split[Sort[First[IntegerDigits[#]]& /@ Flatten[ToExpression /@ allData]]]
```

We see an excellent agreement with the frequencies predicted by Benford's rule.

```
With[{Σ = Plus @@ (Last /@ %) // N},  {#1, #2/Σ}& @@@ %]
```

Next, we implement a function `numberDistribution`. It gives a list of lists with the number of digits in the first *num* digits of the results of the function *func* applied to all integers in *range*. The trivial case (no 0 in the first place) is not included, and only those numbers with enough digits are analyzed.

```
numberDistribution[func_Symbol | func_Function,
                   range_List, num_Integer] :=
If[# != {}, Delete[#, {1, 1}], #]&[(* just counting *)
Table[{k, Count[#, k]}, {k, 0, 9}]& /@  (* make list of digits *)
Transpose[Take[#, num]& /@ IntegerDigits /@
 Select[Table[func[i], Evaluate[Prepend[range, i]]],
          (* select relevant integers *)
          (IntegerQ[#] && (# >= 10^num))&]]]
```

Here are two examples: $\sum_{i=1}^{n}(i+1)$ and $3^n - 2^n$.

```
numberDistribution[Sum[i + 1, {i, #}]&, {1, 100}, 2]

numberDistribution[(3^# - 2^#)&, {1, 200}, 3]
```

To better appreciate the results of `numberDistribution`, we examine the frequencies graphically (the relevant commands are introduced in Chapter 1 of the Graphics volume [301✱]).

```
Needs["Graphics`Legend`"]
```

```
plotNumberDistribution[func_Symbol | func_Function,
                             range_List, num_Integer] :=
Module[{(* the data *) aux = numberDistribution[func, range, num]},
If[aux != {},
ShowLegend[Show[Table[
   ListPlot[aux[[i]],
   (* option setting for a nice plot *)
            PlotJoined -> True, DisplayFunction -> Identity,
            PlotStyle -> {AbsoluteThickness[3],
                        Hue[(i - 1)/num 0.7]}], {i, num}],
     PlotRange -> All, AxesOrigin -> {0, 0},
     DisplayFunction -> Identity,
     AxesLabel -> {"digit", " number of\n occurrences"}],
     (* the legend *)
     {Table[{Graphics[{AbsoluteThickness[2],
      Hue[(i - 1)/num 0.7], Line[{{0, i/num}, {1, i/num}}]}],
            StyleForm["digit" <> ToString[i],
          FontFamily -> "Helvetica", FontSize -> 8]}, {i, num}],
      LegendPosition -> {1.0, -0.4}, LegendSize -> {0.8, 0.4 num/3}}],
           Print["no digits to plot"]]]
```

We now look at three examples: $n!$ [179★], $n+n^2+n^3$, n^n.

```
plotNumberDistribution[Factorial, {1, 250}, 3]
```

```
plotNumberDistribution[(# + #^2 + #^3)&, {1, 1000}, 3]
```

```
plotNumberDistribution[#^#&, {1, 200}, 5]
```

Next, we have a look at the digit distribution for the $3n+1$ problem [168★]. We start at all integers less than 10^4 and carry out the iterations until a cycle is found.

```
threeNPlus1[n0_] := NestWhileList[If[EvenQ[#], #/2, 3 # + 1]&, n0,
                                  (* stopping criteria *)
                                  (# =!= 1 && # =!= 2 && # =!= 4)&]
```

Here are the resulting probabilities for the first digits.

```
Function[l, (* analyze first digit frequencies *)
        {First[#], N @ Length[#]/Length[Flatten[l]]}& /@ l] @
Split[Sort[Flatten[
     (* run 3n + 1 iterations for different starting values *)
         Table[First[IntegerDigits[#]]& /@ threeNPlus1[k],
               {k, 10^4}]]]]
```

For further arithmetical examples see [298★], [111★], [145★], [146★], [337★]; for dynamical system examples, see [282★]; and for discretized images, see [152★]. For deviations from Benford's rule for the weight of crushed stone pieces, see [176★]. .For the first digit, Benford's rule is again more or less satisfied, but not for the later digits.

For an analysis of the probability of appearance of the digit j after the digit i, we have the following distribution p.

```
p[digits_List] :=
With[{n = Length[digits]},
     Log[10, 1 + 1/Sum[digits[[k]] 10^(n - k), {k, n}]]]
```

Summing over all possible values of the second digit we recover the above probabilities for the first digit.

```
Table[Sum[p[{d1, d2}], {d2, 0, 9}], {d1, 1, 9}] -
Table[p[{d1}], {d1, 1, 9}] // Simplify
```

Let us consider the first two digits of $\tan(k\, e/\pi)$ where $1 \le k \le 10^5$.

```
theorData = Table[p[{d1, d2}], {d2, 0, 9}, {d1, 1, 9}];

experData = {First[#], Length[#]}& /@
       Split[Sort[Table[Take[RealDigits[N[Tan[k E/Pi]]][[1]], 2],
                        {k, 10^5}]]];
```

The theoretical probabilities agree quite well with the ones from the sequence $\tan(k\,e/\pi)$.

```
Show[GraphicsArray[
Block[{$DisplayFunction = Identity},
{(* theoretical distribution *)
 ListPlot3D[theorData, MeshRange -> {{1, 9}, {0, 9}},
                       MeshStyle -> {Thickness[0.001]}],
 (* here obtained data *)
 ListPlot3D[Transpose[Map[Last[#]/10^5&,
                      Partition[experData, 10], {2}]],
                PlotRange -> All, MeshRange -> {{1, 9}, {0, 9}},
                MeshStyle -> {Thickness[0.001]}]}]]]
```

The first digits of the powers of 2 [19★] are an example for which it is possible to show analytically that Benford's rule holds. In a few minutes, we can calculate the first digits for 2^k for $k = 1, \ldots, 10^6$.

```
o = 10^6;
data = Table[StringTake[ToString[N[2^k], InputForm], {1, 1}],
             {k, o}];
```

The agreement with the theoretical distribution is excellent.

```
Map[{#[[1]], (* data/theoretical value - 1 *)
             #[[2]]/Log[10, 1 + 1/#[[1]]] - 1}&,
    (* count first digits *)
    {ToExpression[First[#]], N[Length[#]/o]}& /@ Split[Sort[data]]]
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

2. Map, Outer, Inner, and Thread versus Table and Part, Iteratorless Generated Tables, Sum-free Sets

a) First, we create a list with elements.

```
testList = Array[a, 500];
```

We now apply a function `f`, not specified explicitly to each element. We use an inner `Do` loop to get more accurate timings. The function `f` has nontrivial rules in the moment. The `Map` version is much faster than is the `Table[` `Part[...]]]` version.

```
Timing[Do[Map[f, testList], {100}]][[1]]

Timing[Do[Table[f[testList[[i]]], {i, 500}], {100}]][[1]]
```

For comparison, here is a version with a function carrying the attribute `Listable`.

```
SetAttributes[f, Listable];
Timing[Do[f[testList], {100}]][[1]]
```

In addition to the improvement in efficiency, note that in the second construction, the length of the list has to be given explicitly (or a construction like `Length[testMatrix]` has to be used in the iterator). Here is the analogous construction for a matrix.

```
testMatrix = Array[a, {50, 50}];
{Timing[Do[Map[f, testMatrix, {2}], {100}]][[1]],
 Timing[Do[Table[f[testMatrix[[i, j]]], {i, 50}, {j, 50}], {100}]][[1]]}
```

Here are two lists `testLista` and `testListb` that have no values, so that these two lists contain symbolic elements of the form `a[i]`.

```
testLista = Array[a, 500];
testListb = Array[a, 500];
```

We compute the generalized scalar product of the two lists, once using `Inner` and once in the "conventional" way.

```
Timing[Do[Inner[f, testLista, testListb, g], {100}]][[1]]

Timing[Do[g @@ Sum[f[testList[[i]], testList[[i]]],
                  {i, 500}], {100}]][[1]]
```

To test `Outer`, we reduce the size of the matrices somewhat.

```
testLista = Array[a, 50];
testListb = Array[a, 50];

Timing[Do[Outer[f, testLista, testListb], {100}]][[1]]
```

For comparison, here is the conventional approach.

```
Timing[Do[Table[f[testLista[[i]], testListb[[j]]],
                {i, 50}, {j, 50}], {100}]][[1]]
```

For `Thread`, we need some more lists.

```
Do[testListNr[i] = Array[a[i], {20}], {i, 30}]
```

Here, `Thread` is applied.

```
Thread[f @@ Table[testListNr[i], {i, 30}]] // Short[#, 12]&

Timing[Do[Thread[f @@ Table[testListNr[i], {i, 30}]], {100}]][[1]]
```

Here, all equivalent elements are individually identified and further applied. The savings in time is significant, because a conventional approach requires operating 20 times on 30 different lists.

```
Timing[Do[Table[f @@ Table[testListNr[i][[j]], {i, 30}], {j, 20}], {100}]][
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

b) `Outer` gives the possibility to create such a table. Because all *m*-ranges for the iterators are the same, we generate just one of them, make a table of them, and apply `Outer[f, ##]&` to this table.

```
functionalTableMaker[f_, n_, m_] :=
         Outer[f, ##]& @@ Table[#, {n}]&[Range[m]]
```

Let us check the equivalence of the result of `functionalTableMaker` with the `Table` version and compare timings.

```
functionalTableMaker[ABC, 4, 5] ===
Table[ABC[i1, i2, i3, i4], {i1, 1, 5}, {i2, 1, 5}, {i3, 1, 5}, {i4, 1, 5}]

Timing[Do[Table[ABC[i1, i2, i3, i4, i5, i6],
                {i1, 1, 4}, {i2, 1, 4}, {i3, 1, 4},
                {i4, 1, 4}, {i5, 1, 4}, {i6, 1, 4}], {100}]]

Timing[Do[functionalTableMaker[ABC, 6, 4], {100}]]
```

As expected, the functional version is much faster. Another possibility is the use of `Array`.

```
functionalTableMaker2[f_, n_, m_] := Array[f, Array[m&, n]]
```

```
functionalTableMaker2[ABC, 6, 4] ===
Table[ABC[i1, i2, i3, i4, i5, i6],
      {i1, 1, 4}, {i2, 1, 4}, {i3, 1, 4},
      {i4, 1, 4}, {i5, 1, 4}, {i6, 1, 4}]

Timing[Do[functionalTableMaker2[ABC, 6, 4], {100}]]
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

c) Here are a couple of functional and procedural programmed possibilities to perform the task. They should be reviewed in detail to see how the various constructs work.

```
m[1][f_, mat_] := Transpose[MapThread[#2 /@ #1&, {Transpose[mat], f}]]

m[2][f_, mat_] := Inner[#2[#1]&, mat, f, List]

m[3][f_, mat_] := MapThread[#1[#2]&, {f, #}]& /@ mat

m[4][f_, mat_] := Module[{mat1 = mat},
                        Do[mat1[[i]] = Inner[#2[#1]&, mat[[i]], f, List],
                         {i, 1, Length[f]}]; mat1]

m[5][f_, mat_] := Table[f[[j]][mat[[i, j]]], {i, Length[f]}, {j, Length[f]}

m[6][f_, mat_] := MapIndexed[f[[#2[[2]]]][#1]&, mat, {2}]

m[7][f_, mat_] := Module[{mat1 = mat},
                        Do[mat1[[i, j]] = f[[j]][mat[[i, j]]],
                           {i, Length[f]}, {j, Length[f]}]; mat1]

m[8][f_, mat_] := Module[{mat1 = mat, matHold = Hold @@ {mat},
                          (* avoid evaluation *)
                          fHold = Hold @@ {f}},
                          Do[mat1[[i, j]] = fHold[[1, j]][matHold[[1, i, j]]
                             {i, Length[f]}, {j, Length[f]}]; mat1]
```

Let us test that all `m[i]` really generate the same result.

```
SameQ @@ (Function[{f, mat}, #[f, mat]& /@ Table[m[i], {i, 8}]][
                    Array[k, 5], Array[b, {5, 5}]])
```

Now, let us time the various programming constructs with differently sized matrices.

```
(* format timing uniformly *)
timeString[t_Real, afterCommaDigits_] :=
Module[{τ = ToString[t], p, σ},
       (* smaller than display granularity *)
       If[t < 10^-afterCommaDigits, "0." <>
                StringJoin[Table["0", {afterCommaDigits}]],
       (* format nicely string *)
       p = StringPosition[τ, "."][[1, 1]];
       σ = StringTake[τ, {1, Min[p + afterCommaDigits, StringLength[τ]]}];
       If[StringLength[σ] < p + afterCommaDigits,
          StringJoin[σ, Table["0", {(p + afterCommaDigits) -
                                     StringLength[σ]}]], σ]]]
```

```
With[{minDim = 20, maxDim = 200, stepDim = 8},
Module[{timings, testMat, testf},
(* get timings *)
timings = Table[testMat = Array[b, {dim, dim}]; testf = Array[k, dim];
                Table[Timing[m[i][testf, testMat]][[1, 1]], {i, 8}],
                {dim, minDim, maxDim, stepDim}];
(* format results *)
TableForm[Map[timeString[#, 2]&, timings, {-1}], TableHeadings ->
          {Table[dim, {dim, minDim, maxDim, stepDim}],
                Table[ToString[m[i]] <> "\n\n", {i, 8}]}]]]
```

The first method, which always treats one column (or row) at once, is the fastest.

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

c) We start by implementing the procedural approach. To do this, we operate with the two lists `set` and `sums`. The function `next` returns the smallest integer that is larger than the largest element of set and that is not contained in *sums*.

```
next[set_, sums_] :=
Module[{max = Last[set], pos, new, l = Length[sums]},
  (* position of largest sum smaller than largest element *)
 pos = Position[sums, _?(# > max&), {1}, 1][[1, 1]];
  (* find next integer that is not an already encountered sum *)
 While[new = sums[[pos]] + 1;
       pos = pos + 1; pos <= l && sums[[pos]] == new,
       Null];
 new]
```

The function `update` adds the integer *new* to the list *set* and adds all sums that can be formed using *set* and *new* to the list *sums*. It returns the updated lists *set* and *sums*.

```
update[set_, sums_, new_] :=
  (* add element and all new sums *)
  {Append[set, new], Union[Flatten[{sums, new + set}]]}
```

Using the two functions `next` and `update`, it is straightforward to implement the function `enlargeSetProce⋱dural` that adds *n* integers to the initial list *initialSet*.

```
enlargeSetProcedural[initialSet_, n_] :=
Module[{set = initialSet, sums, new},
       sums = Union[Flatten[Outer[Plus, set, set]]];
       Do[new = next[set, sums];
       {set, sums} = update[set, sums, new], {n}];
       set]
```

Here is a simple example showing how `enlargeSetProcedural` works.

```
enlargeSetProcedural[{1, 2, 3, 4, 5}, 6]
```

Now let us implement the function `enlargeSetUsingCaching`. Instead of using a list to store all elements and all sums, we define functions `element` and `isSumQ` which contain the information.

```
enlargeSetUsingCaching[initialSet_, n_] :=
Module[{element, isSumQ, l = Length[initialSet], counter, max},
(* initialize with given numbers *)
MapIndexed[(element[#2[[1]]] = #1)&, initialSet];
counter = l;
(* initialize isSumQ with all sums
  that can be formed from initialSet *)
 (isSumQ[#] = True)& /@  Union[Flatten[
                Outer[Plus, initialSet, initialSet]]];
(* add n elements to the set *)
Do[With[{max = element[counter]},
    (* starting at the largest element in the set find the
      smallest integer that is not an already formable sum *)
        For[k = 1, isSumQ[max + k], k = k + 1, Null];
            element[counter = counter + 1] = max + k;
            (* add new possible sums *)
            Do[isSumQ[element[j] + element[counter]] = True,
               {j, counter}]], {n}];
(* return all elements *)
Table[element[k], {k, l + n}]]
```

Again, we use the simple starting sequence {1, 2, 3, 4, 5} for a quick check of `enlargeSetUsingCaching`.

```
enlargeSetUsingCaching[{1, 2, 3, 4, 5}, 6]
```

Now let us compare the timings of `enlargeSetProcedural` and `enlargeSetProcedural` for the starting set being the first ten primes for 2000 recursive enlargements.

```
primeSet = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29};
```

```
(𝒮P2000 = enlargeSetProcedural[primeSet, 2000];) // Timing
```

```
(𝒮C2000 = enlargeSetUsingCaching[primeSet, 2000];) // Timing
```

The timings are nearly identical and the two calculated sets agree too.

```
𝒮P2000 === 𝒮C2000
```

The complexity of both implementations is about $O(n^2)$ for small n. `enlargeSetUsingCaching` will be asymptotically faster. `enlargeSetProcedural` will have to search through larger and larger lists, whereas `enlargeSetUs`⋱`ingCaching` does not have to do this. But both functions will create at each step the constantly increasing sets of new sums.

```
{#, Timing[enlargeSetProcedural[primeSet, #][[1]]]}& /@
                                         {100, 200, 400, 800, 1600}
```

```
{#, Timing[enlargeSetUsingCaching[primeSet, #][[1]]]}& /@
                                         {100, 200, 400, 800, 1600}
```

Interestingly, the sequence of differences between the numbers n_k it seems to become periodic for any starting set $\mathcal{S}_o$ [86✱], [205✱].

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

3. Index

The list of the commands introduced in this book is `Private`IntroducedCommands` in the package `Chapter`Overview`. The form for each chapter is `{"`*command*`", "`*whereIntroduced*`"}`, where *whereIntroduced* is the section of subsection in form of a string where the command *command* was discussed. There a consecutive numbering (ranging from 1 to 14) of all chapters is used. We create a list of the same names in the context `Global``. (In addition, we assume that the list of all built-in commands comes from this package.)

```
Get[ToFileName[ReplacePart[
            "FileName" /. NotebookInformation[EvaluationNotebook[]],
            "ChapterOverview.m", 2]]];

introducedCommandsPre =
   GuideBooks`ChapterOverview`Private`IntroducedCommands;

introducedCommands = Union[
   Flatten[Map[First, introducedCommandsPre, {2}]]];

allCommands = Names["System`*"];
```

Here is a shortened version of the list of commands introduced in boxes in this book.

```
introducedCommands // Short[#, 14]&
```

This is the total number of commands.

```
Length[introducedCommands]
```

Here is the *Mathematica* index. The function `whereIntroduced` gives the section in which the command was introduced.

```
whereIntroduced[command_] :=
Module[{aux},
       (* where it is *)
       aux = Position[introducedCommandsPre, command];
       (* return chapter and section numbering *)
       If[aux == {}, "This command was not introduced.",
          consecutiveNumberingToPartNumbering /@
            (Part[introducedCommandsPre, #[[1]], #[[2]], 2]& /@ aux)]]

(* convert from consecutive to four-volume numbering *)
consecutiveNumberingToPartNumbering[s_String] :=
Module[{spos = StringPosition[s, ".", 1][[1, 1]], cn, rest},
        cn = ToExpression[StringTake[s, {1, spos - 1}]];
        rest = StringTake[s, {spos, StringLength[s]}];
       (* 6 Programming, 3 Graphics,
    2 Numerics, and 3 Symbolics chapters *)
       Which[cn <  7, "P_" <> ToString[cn],
             cn < 10, "G_" <> ToString[cn - 6],
             cn < 12, "N_" <> ToString[cn - 9],
             cn < 15, "S_" <> ToString[cn - 11]] <> rest]
```

Here are a few examples involving commands that were introduced just once.

```
whereIntroduced["TableForm"]

whereIntroduced["InputForm"]
```

We have mentioned `Plot` twice, once at the beginning of Chapter 3 to plot something and again in more detail in Chapter 1 of the Graphics volume [301✱].

```
whereIntroduced["Plot"]
```

The following command was not treated in this book at all.

```
whereIntroduced["PolynomialMod"]
```

Here are all the commands that were not discussed.

```
Complement[allCommands, introducedCommands] // Short[#, 16]&
```

Many commands were not introduced.

```
Length[Complement[allCommands, introducedCommands]]
```

Did we misspell the name of any command in `introducedCommands`; that is, is there a command in our list that does not appear in the list produced by `Names["*"]`?

```
Complement[introducedCommands, allCommands]
```

How many *Mathematica* commands were introduced in the various chapters?

```
Do[CellPrint[Cell["∘ In Chapter " <>
  Which[i <  7, "Programming_" <> #[i],
        i < 10, "Graphics_" <> #[i - 6],
        i < 12, "Numerics_" <> #[i - 9],
        i < 15, "Symbolics_" <> #[i - 11]]&[ToString] <>
                 ", a total of " <>
                  ToString[Length[introducedCommandsPre[[i - 1]]]] <>
                  " commands were discussed.", "PrintText"]],
   {i, 2, 14}]
```

Which commands were discussed more than once, and in which sections? Here are the reasons for the multiple appearances.

■ Their operations depend on their argument.

■ They are used for both 2D and 3D graphics.

■ They are first introduced, and then later discussed in detail.

```
({#[[1, 1]], (* where it appeared? *)
                 whereIntroduced[#[[1, 1]]]}& /@
  Select[Split[Sort[Flatten[introducedCommandsPre, 1]],
               #1[[1]] === #2[[1]]&],
            (* at least two times mentioned *) Length[#] > 1&]) //
                                                    Short[#, 12]&
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

4. Functions Used Too Early?, Check of References, Closing]], Line Lengths, Distribution of Initials, Check of Spacings

a) The idea is the following: After reading in the notebooks and extracting the inputs as well all definitions of *Mathematica* commands, we decompose each *Mathematica* input with `Level[..., {-1}, Heads -> True]` into its basic parts, and pick out all built-in commands that are used there but that have not yet been introduced. We begin with the built-in commands. For later use, we enclose them in `Hold`.

```
allSystemCommands = ToHeldExpression /@ Union[Names["System`*"]];
```

The list `alreadyIntroducedCommands` needs to be updated. Here is what it looks like at the end of Chapter 2. (The commands are collected as strings, analogous to the list `introducedCommands` in the previous problem.)

```
alreadyIntroducedCommands =
{"FullForm", "TreeForm", "InputForm", "OutputForm", "Head", "Integer",
 "Rational", "Real", "Complex", "String", "Plus", "Times", "Power",
 "Sqrt", "Symbol", "Subtract", "Divide", "Minus", "Exp", "Sin", "Cos",
 "Tan", "Cot", "Sec", "Csc", "Sinh", "Cosh", "Tanh", "Coth", "Sech",
 "Csch", "N", "I", "Pi", "Degree", "E", "GoldenRatio", "EulerGamma",
 "DirectedInfinity", "ComplexInfinity", "Indeterminate", "ArcSin",
 "ArcCos", "ArcTan", "ArcCot", "ArcSec", "ArcCsc", "ArcSinh", "ArcCosh",
 "ArcTanh", "ArcCoth", "ArcSech", "ArcCsch", "Re", "Im", "Arg", "Abs",
 "N", "Short", "Shallow", "Skeleton", "Part", "Depth", "Position",
 "Level", "Heads", "Length", "LeafCount", "Numerator", "Denominator",
 "IntegerDigits", "RealDigits", "BaseForm"};
```

The main programming work is in searching for the commands of the inputs that have not yet been used. The function `orderCheck` does this task. Its operation is more or less analogous to that in Section 6.6. First, we enclose all atomic subexpressions in unevaluated form (attribute `HoldAll` in `Function`) in `Hold`. Next, we extract the commands that have already been introduced. The remaining `Hold[`*var*`]` are analyzed to see if they are built-in functions. `Rest` is needed to remove the first `Hold[Hold]`. If the resulting list is not `{}`, it is printed. The argument is returned unchanged. For better readability, in particular for expressions containing many inputs, we print the `In[...]` numbering and the analyzed expression in the input form.

```
orderCheck :=
Function[x, (* print the result of the analysis *)
          Function[y, If[y != {},
CellPrint[Cell[TextData[{
  StyleBox["∘ In "],
  StyleBox["In[" <> ToString[$Line] <> "]", "CellLabel"],
  StyleBox["\n"],
  StyleBox[StringDrop[StringDrop[ToString[
        InputForm[Unevaluated[x]]], 12], -1], FontFamily -> "Courier"],
  StyleBox[
   "\n∘ The following, until now not discussed, functions were used: \n"],
  StyleBox[ToString[y], FontFamily -> "Courier"]}], "PrintText"]]]][
          (* wrap Hold around everywhere and then look
      if it was already introduced *)  (HoldForm @@ #)& /@
         Intersection[Complement[Rest[Level[
                  Map[Hold, Hold[x], {-1}],
                      Heads -> True], {-2}, Heads -> True]],
          (* introduced commands *)
          ToHeldExpression /@ alreadyIntroducedCommands],
                              allSystemCommands]]; x, {HoldAll}]
```

Here, we check an example expression for new commands (we see that no subexpression is computed, and `Hold` appears exactly once).

```
orderCheck[y[x_] := Block[{$RecursionLimit = 100, v},
                          $IterationLimit; Hold;
                          Blank; Blank; Blank;
                          Unevaluated[Integrate];
                          $Version; Date;
                          v[y_] := v[y - 1]2 + 3;
                          v[0] = 456;
                          v[x]]]
```

If no new command appears, nothing is printed.

```
Sin[x^3] + Cos[x y] + 12 // orderCheck
```

Now, if we set `$Pre = orderCheck`, every *Mathematica* input will automatically be checked for commands that have not yet been introduced (note that the expression given to `$Pre` must be a function). We do not discuss the test, but the interested reader can see how frequent commands that were not introduced were actually used.

Note that the given function is applied to every *Mathematica* input. Thus, the following approach will not work because we cannot wrap `orderCheck` around multiple expressions. After doing so multiple expressions will be interpreted as factors of a product. The warning message `RuleDelayed::rhs` is issued because of the `(fu1[x_] := Sin[x], x^2)*(fu2[x_] := Cos[x], x^3)` interpretation of the argument.

```
orderCheck[fu1[x_] := {Sin[x], x^2}
           fu2[x_] := {Cos[x], x^3}]
```

However, the following example does work.

```
$Pre = orderCheck;

fu1[x_] := {Sin[x], x^2}
fu2[x_] := {Cos[x], x^3}
```

The analysis of the results of calculations with *Mathematica* can be done in a similar way. In this case, no additional work is needed to prevent the computation of the parts.

Once in a while, we use commands that have not yet been "officially" introduced. To analyze these cases correctly, we could introduce a variable `$orderCheck`, with possible values `True` and `False`, which tells whether to check the following input:

```
$Pre = If[$orderCheck == True, orderCheck, Identity]
```

To prevent it from checking itself, the two inputs `$orderCheck = True` and `$orderCheck = False` have to be treated specially. We do not give further details here. We conclude by recovering the original value of `$Pre`.

```
$Pre =.
```

Evaluating `$Pre` now has no side effect.

```
$Pre =.
```

Anyway, we leave it to the reader to check how often we used commands that were not introduced at all or how often commands were used before they were "officially" introduced. It happened sometimes. So the reader did not really expect to find the actual answer to the posed question here. To really figure out how often it happens that commands are used before they are explained, we must read in the notebooks forming the *GuideBooks*, extract the cells (and their positions) that introduce new commands (they are of type `"MathDescription"`) and compare these with the actual commands used in cells of type `"Input"`. The list of functions introduced before a certain `"Input"` cell must be updated after every occurrence of a `"MathDescription"` cell.

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

b) We do not carry out this check by hand; this would be too time-consuming and error-prone. Because notebooks are *Mathematica* expressions, we can carry out the check inside *Mathematica*. The references in the Reference sections are in cells of the type `Cell[`*referenceDetails*`, "BibliographyItem", CellTags -> "`*LastNameOfTheFirstAuthorAndTwoDigitYear*`"]`. We extract these cells and then extract the relevant *LastNameOfTheFirstAuthorAndTwoDigitYear* from the cells. This process gives us the list of references. In the main text of the *GuideBooks* chapters, we refer to a reference in the form `ButtonBox["✱", ButtonData :> "`*LastNameOfTheFirstAuthorAndTwoDigitYear*`", ButtonStyle -> "Hyperlink"]` (this is the underlying expression *Mathematica* expression in the notebook). We extract the right-hand side of the `ButtonData` option, and this gives us a list of all the references we refer to. Then, we just compare if any elements are in the referred references that are not in the listed references and opposite.

```
notebooks = Flatten[
     {Function[{c, n}, (c <> ToString[#] <> ".nb")& /@ Range[n]] @@@
      {{"1_Programming_", 6}, {"2_Graphics_", 3},
       {"3_Numerics_", 2}, {"4_Symbolics_", 3}},
      "Preface.nb", "0_Introduction.nb", "Appendix_A.nb"}]

(* directory name *)
dirName = StringDrop[ToFileName["FileName" /.
            NotebookInformation[EvaluationNotebook[]]], -18];
```

Here is an implementation of the program sketched above.

```
Function[nbs,
(* read in a notebook *)
nb = Get[(* construct file name *) dirName <> nbs];
(* check tags for unresolved counter boxes *)
If[MemberQ[nb, CounterBox["BibliographyCounter", _], Infinity],
   If[# =!= {}, Print["Different tags in " <> nbs <> " : ", List @@@ #]]& @
     Select[Union /@ Cases[nb //. (* extract counter structures *)
     {a___, CounterBox["BibliographyCounter", r_],
            ButtonBox["✱",  ButtonData :> r_, __], b___} :>
                {a, C[r, r], b}, _C, Infinity], Length[#] === 2&]];
(* the references *)
references = Flatten[Last /@
 Cases[Cases[nb, Cell[__, "BibliographyItem", __], Infinity],
       HoldPattern[CellTags -> r_], Infinity]];
(* are tags unique? *)
If[Not[Sort[references] === Union[references]],
   Print["Multiple tags in file: ", nbs];
   Print[Select[Split[Sort[references]], (Length[#] >= 2)&]]];
(* the mentioned references *)
referredToReferences = Flatten[#[[2, 2]]& /@
    Cases[nb, ButtonBox["✱", ButtonData :> _,
              ButtonStyle -> "Hyperlink", ___], Infinity]];
(* analyse all data *)
{nbs, {(* how many entries? *) Length /@ #,
         (* any entries unused or unreferenced? *)
   (* any entries unused? *) Complement @@ #,
         (* any entries unreferenced? *)
         Complement @@ Reverse[#]}}&[
           {references, referredToReferences}]] /@ notebooks
```

Luckily, all second arguments of the last lists are empty, which means each mentioned reference is really present and each given reference is mentioned at least once.

This input gives the total number of references (counted with their multiplicity in case they are used in more than one chapter).

```
Plus @@ (#[[2, 1, 1]]& /@ %)
```

Here are the current number of *Arxiv*-, DOI-, book-, and direct hyper-links.

```
Function[refs, Count[refs, ButtonStyle -> #, {-2}]& /@
  (* link types *) {"ArXivLink", "DOILink", "BookLink", "Hyperlink"}][
   Flatten[Table[Cases[Get[dirName <> notebooks[[k]]],
                        Cell[___, "BibliographyItem", ___], Infinity],
                 {k, Length[notebooks]}]]]
```

This means that about 73% of all refrences are hyperlinked.

```
N[(Plus @@ %)/%%]
```

Now, let us analyze which are the most-cited journals. We extract all italic journal (and book) titles from the references.

```
data = Table[
 (* the notebook to be analyzed *)
 nb = Get[dirName <> notebooks[[k]]];
 (* the journal and book titles in the reference cells *)
 items = First /@ Cases[Cases[nb, Cell[___, "BibliographyItem", ___],
                         Infinity], StyleBox[_, "TI"], Infinity],
                 {k, Length[notebooks]}];
```

We sort the titles and count how frequently they occur.

```
res = Sort[Split[Sort[Flatten[data]]], Length[#1] > Length[#2]&];
```

Here are the 12 most cited journals. As mentioned in the introduction, most examples come from general physics, mathematics, and related fields [206✱] (and, of course, *Mathematica*-related journals). Five of the arXiv physics preprint groups made it into the top ten.

```
(* format nicely *)
GridBox[{StyleBox[First[#], FontFamily -> "Times", FontSlant -> Italic],
         Length[#]}& /@ Take[res, 12],
        ColumnAlignments -> {Left, Right}] // DisplayForm
```

The function `publicationYear` extracts the year of the publication of a journal article or a book from a `Bibliog`↳ `raphyItem` cell.

```
publicationYear[ref_] :=
Module[{ref1 = DeleteCases[ref, _ButtonBox, Infinity],
        str, sp1, sp2, year},
(* extract journals and preprints; no books *)
str = Cases[ref, _String?(StringMatchQ[#, "*(*)*"]&), {-1}];
If[str =!= {},
   (* a journal or preprint citation *)
   {sp1, sp2} = StringPosition[str[[-1]], #]& /@ {"(", ")"};
   year = StringTake[str[[-1]], {sp1[[-1, 1]] + 1, sp2[[-1, 1]] - 1}]];
If[Head[year] === String && SyntaxQ[year] &&
   (* excluding the publication year of this book *)
   TrueQ[1600 <= ToExpression[year] <= 2004], Null,
   (* a book citation *)
   str = Cases[ref2, TextData[{___, r_}] :> r];
   If[str =!= {},
      sp2 = StringPosition[str[[-1]], "."];
      year = StringTake[str[[-1]], {sp2[[-1, 1]] - 4, sp2[[-1, 1]] - 1}]]];
If[Head[year] === String && SyntaxQ[year] &&
   (* until the completed last year *)
   TrueQ[1600 <= ToExpression[year] <= 2004], year]]
```

We read in all notebooks and determine the publication years of all citations. We separately count the electronic articles. They either refer to a URL (visible in the `ButtonFunction` as `URL`) or have a “Get Preprint” button.

```
data = Table[
nb = Get[(* construct file name *) dirName <> notebooks[[k]]];
(* the references *)
references = Cases[nb, Cell[__, "BibliographyItem", __], Infinity];
(* the references to electronic documents *)
eReferences = Select[references,
                     (MemberQ[#, "Get Preprint", {-1}] ||
                      MemberQ[#, URL, {-1}, Heads -> True])&];
(* the publication years *)
DeleteCases[{publicationYear /@ eReferences,
             publicationYear /@ references}, Null, {2}],
            {k, Length[notebooks]}];
```

Here is the number of electronic articles over the last ten years.

```
eData = Select[{ToExpression[First[#]], Length[#]}& /@
                          Split[Sort[Flatten[First /@ data]]],
               #[[1]] <= 2004&]
```

In a logarithmic plot, the exponential increase of electronic articles in the nineties becomes easily visible.

```
ListPlot[aux = Apply[{#1, Log[10, #2]}&, eData, {1}],
         PlotJoined -> True, Frame -> True, Axes -> False,
         Epilog -> {PointSize[0.02], Point /@ aux},
         FrameTicks -> {Table[j, {j, 1988, 2004, 2}],
                        Automatic, None, None}]
```

The number of electronic articles nearly exactly doubles from year to year. This is in agreement with general estimations. (See the electronic articles [230✱], [187✱], [245✱]; for printed literature, see [16✱]). For the arXiv statistics, see http://arXiv.org/cgi-bin/show_monthly_submissions.) And the "starting date" of electronic articles (mentioned in this book) is in 1991 [21✱].

```
With[{fit = (* take data from 1992 to 2000 and extrapolate *)
            Fit[Cases[aux, {_?(1992 <= # <= 2000&), _}], {1, x}, x]},
     {10^Coefficient[fit, x, 1], x /. Solve[fit == 0, x][[1]]}]
```

Now let us see what fraction the electronic articles constitute among all references.

```
allData = Select[{ToExpression[First[#]], Length[#]}& /@
                 Split[Sort[Flatten[Last /@ data]]],
                 (* use =only full years *) #[[1]] <= 2004&];

(* select relevant years *)
allData1 = Cases[allData, Alternatives @@ ({#, _}& /@
                                          (First /@ eData))];

eFraction = MapThread[{#1[[1]], #1[[2]]/#2[[2]]}&, {eData, allData1}];
```

The relative fraction of electronic articles reached about 50% in 2000 (this distribution is not unique because there a preprint may appear in a journal later). (The relatively steep increase in the number of references in the years 1998–2000 is largely caused by the new symbolic and numeric computing capabilities that precipitated with the release of Version 4.0 of *Mathematica*.)

```
ListPlot[eFraction, PlotJoined -> True, Frame -> True, Axes -> False,
         Epilog -> {PointSize[0.02], Point /@ eFraction},
         FrameTicks -> {Table[j, {j, 1988, 2004, 2}],
                        Automatic, None, None}]
```

Plotting the total number of references as a function of their age in a double logarithmic plot shows clearly two different distribution regimes [123✱]. The number of cited references decreases more quickly for references that are older than about $10^{1.05} \approx 11$ years.

```
logLogData = {Log[10, 2005 - ToExpression[First[#]]],
              Log[10, Last[#]]}& /@ allData;

ListPlot[logLogData, PlotRange -> All, Frame -> True, Axes -> False]
```

Here are the approximate decay powers for the two regimes.

```
Function[lg, Fit[Select[logLogData, (#[[1]] ~ lg ~ 1.05)&],
                 {1, x}, x]] /@ {Less, Greater}
```

To quantify the crossover point, we calculate the weighted residue for a set of linear fits for the citation counts between the ages *ageList*.

```
residue[ageList_List] :=
Module[{ageRanges, rawData, rangeData, fitFunctions,
        (* ignore very recent papers *) minLogAge = Log[10, 2.5]},
 If[OrderedQ[ageList],
    (* age intervals *)
    ageRanges = Partition[Flatten[{minLogAge, ageList, Infinity}], 2, 1];
    (* citation data *)
    rawData = N @ Select[logLogData, #1[[1]] >= minLogAge&];
    (* citation data in age intervals *)
    rangeData = Function[{age1, age2},
                      Select[rawData, (age1 <= #[[1]] <= age2)&]] @@@
                                                              ageRanges;
    (* linear fits to citation data in age intervals *)
    fitFunctions = Function[x, Evaluate[Fit[#, {1, x}, x]]]& /@ rangeData;
    (* sums of squares of differences; weighted by citation counts *)
    Sum[(Plus @@ (Evaluate[#2 Abs[fitFunctions[[k]][#1] - #2]]& @@@
               rangeData[[k]])), {k, Length[ageRanges]}],
  Infinity]]
```

The left graphic shows the residue as a function of one crossover age and the right 3D graphic shows the residue as a function of two crossover ages.

```
Show[GraphicsArray[
   Block[{$DisplayFunction = Identity},
      {(* residue for fit with two linear functions *)
       Plot[residue[{age}], {age, 0, 2}, AxesLabel -> {"age", None}],
       (* residue for fit with three linear functions *)
       Plot3D[residue[{age1, age2}], {age1, 0, 2}, {age2, 0, 2},
              PlotPoints -> 160,  AxesLabel -> {"age1", "age2", None},
              ViewPoint -> {-3, -0.8, 2}, Mesh -> False]}]]] //
                                           Internal`DeactivateMessages
```

For one crossover point, we find again the age of about 12 years and for two crossover points, we find the second age to be about 45 years.

```
Module[{allPoints, minPoints},
 (* extract points from all polygons *)
 allPoints = Cases[Level[Cases[Graphics3D[%[[1, 2]]],
                                  _Polygon, Infinity], {-2}], _List];
 (* smallest residue values *)
 minPoints = Union[Cases[allPoints, {_, _, Min[Last /@ allPoints]}]];
 10^(Most /@ minPoints) "years"]
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

c) These are the notebooks to be analyzed.

```
notebooks = Flatten[
     {Function[{c, n}, (c <> ToString[#] <> ".nb")& /@ Range[n]] @@@
      {{"1_Programming_", 6}, {"2_Graphics_", 3},
       {"3_Numerics_", 2}, {"4_Symbolics_", 3}},
      "Preface.nb", "0_Introduction.nb", "Appendix_A.nb"}];

fileNames = ToFileName[ReplacePart["FileName" /.
  NotebookInformation[EvaluationNotebook[]], #, 2]]& /@ notebooks;

Do[nb[k] = Get[fileNames[[k]]], {k, 17}]
```

We extract all reference cells.

```
bibliographyItems[nb_] :=
             Cases[nb, Cell[___, "BibliographyItem", ___], Infinity]
```

The part *referenceCell*`[[1, 1, 3]]` is the string of the author names. The function `getLetters` analyzes the string and extracts the first letters of the initial, middle, and the last names.

```
getLetters[s_String] :=
Module[{chars = Characters[s], upperCasePosis, initialAndMiddleNamePosis},
(* position of upper case letters *)
upperCasePosis = Flatten[Position[chars, _?UpperCaseQ, {1}, Heads -> False]
(* position of upper case letters of initials;
  all initial and middle names are abbreviated *)
initialAndMiddleNamePosis = Select[upperCasePosis, (chars[[# + 1]] === ".")
{(* first letter of initials *)
 chars[[initialAndMiddleNamePosis]],
  (* first letter of lastname *)
 chars[[Complement[upperCasePosis, initialAndMiddleNamePosis]]]}]

getLetters[StyleBox[s_String, ___]] := getLetters[s]

(* extract the names from a reference cell *)
extractNameString[Cell[TextData[l_], ___]] :=
With[{pos = Position[l, _String, {1}, 1]},
     If[pos =!= {}, l[[pos[[1, 1]]]]]]
```

Now, we extract all first letters and count their appearance.

```
allLetters = Flatten[Table[getLetters[extractNameString[#]]& /@
                         bibliographyItems[nb[k]], {k, 17}], 1];
```

So the most frequent first and middle names start with J and the most frequent last names start with S.

```
Take[Sort[{First[#], Length[#]}& /@ Split[Sort[Flatten[First /@ allLetters]
                                   #1[[2]] > #2[[2]]&], 10]

ReplacePart[DownValues[In][[-2]], Last, {2, 1, 1, 2, 1, 1, 1, 1}][[2]]
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

d) Now, let us analyze the line lengths of the inputs and the relative fraction of white space. The function `lines` splits a string containing newline characters into single lines.

```
lines[s_String] := StringTake[s, #]& /@
      Partition[Flatten[{1, StringPosition[s, "\n"], StringLength[s]}], 2]
```

We read in all chapters, select the inputs, split the inputs into lines, and determine the lengths of the lines as well as the number of white space characters.

```
notebooks = Flatten[
     {Function[{c, n}, (c <> ToString[#] <> ".nb")& /@ Range[n]] @@@
      {{"1_Programming_", 6}, {"2_Graphics_", 3},
       {"3_Numerics_", 2}, {"4_Symbolics_", 3}}}];

fileNames = ToFileName[ReplacePart["FileName" /.
  NotebookInformation[EvaluationNotebook[]], #, 2]]& /@ notebooks;

(* indentation of a Mathematica input line *)
indentation[s_String] :=
With[{chars = Characters[s]},
 If[(* no nontrivial characters on this line *)
    StringLength[s] <= 1 ||
          Complement[chars, {"\n", "\t", " "}] === {}, 0,
    Position[Rest[chars], _?(# =!= " "&), {1}, 1,
             Heads -> False][[1, 1]]]]

data = Table[
 nb = Get[fileNames[[k]]];
 (* get input cells *)
 inputCells = Cases[nb, Cell[__, "Input", ___], Infinity];
 (* the input strings *)
 inputStrings =  Which[Head[#[[1]]] === String, #[[1]],
                       Head[#[[1]]] === TextData,
                       Check[StringJoin @@ DeleteCases[#[[1, 1]],
                                 _StyleBox], Sequence @@ {}],
                       True, Sequence @@ {}]& /@ inputCells;
 (* split input cells into individual lines *)
 allLines = Flatten[lines /@ inputStrings];
 (* get line lengths and count white spaces *)
 {StringLength[#], Count[Characters[#], " "],
  indentation[#]}& /@ allLines, {k, 1, 14}];
```

The next graphic shows the distribution of the line lengths. The peak for short line length is caused by inputs like `%`, `N[%]`, ….

```
ListPlot[{First[#], Length[#]}& /@
              Drop[Split[Sort[First /@ Flatten[data, 1]]], 1],
        Frame -> True, Axes -> False, PlotJoined -> True,
        PlotRange -> {{0, 80}, All}]
```

About one-fifth of the inputs are white space.

```
N[#2/#1& @@ Apply[Plus, Transpose[Flatten[data, 1]], {1}]]
```

In average, the inputs are indented by about five to six characters.

```
indents = {First[#], Length[#]}& /@
            Split[Sort[Flatten[Last /@ Flatten[data, 1]]]];

(Plus @@ (Times @@@ indents))/(Plus @@ (Last /@ indents)) // N
```

We end with analyzing the density of code comments. For a given cell of type `"Input"` or `"Program"`, the function `commentAndCodeLines` counts the number of lines of comments and code.

```
(* count number of newline characters in an expression *)
countNewlineChars[expr_] := Length @
  StringPosition[StringJoin[Cases[expr, _String, {-1}]], "\n"]

commentAndCodeLines[inputAndProgramCell_] :=
Module[{numberOfCommentLines, s1, s2},
 If[FreeQ[inputAndProgramCell, _BoxData, Infinity],
 {(* count number of comment lines *)
  numberOfCommentLines = Plus @@ ((1 + countNewlineChars[#])& /@
  Cases[inputAndProgramCell, StyleBox[_, "CodeComment", ___],
        Infinity]),
 (* count number of code lines *)
 s1 = DeleteCases[inputAndProgramCell[[1]],
                  StyleBox[_, "CodeComment", ___], Infinity] /.
                  StyleBox[x_, __] :> x;
 If[(* comment only case *) s1 === TextData[], 0,
    (* ignore empty lines *)
    s2 = If[Head[s1] === String, s1, StringJoin[s1[[1]]]];
    Plus @@ (If[Complement[Union[Characters[#]], {"\n", " "}] =!= {},
                1, 0]& /@ (StringTake[s2, #]& /@ Partition[
    Union[Flatten[{1, First /@ StringPosition[s2, "\n"],
                  StringLength[s2]}]], 2, 1]))]}, Sequence @@ {}]]
```

Extracting now the input and program cells for all 14 chapter notebooks yields the following counting data.

```
data = Table[
  (* load notebook *) nb = Get[fileNames[[j]]];
  (* extract input and program cells *)
  inputAndProgramCells =
    Cases[Flatten[nb[[1]] //. Cell[CellGroupData[l_, ___], ___] :> l],
          Cell[_, "Input" | "Program", ___]];
  (* analyze inputs and comments *)
  Table[commentAndCodeLines @ inputAndProgramCells[[k]],
       {k, Length[inputAndProgramCells]}], {j, 14}];
```

On average, we have one comment per six lines of code.

```
Divide @@ (Plus @@@ Transpose[Flatten[data, 1]]) // N
```

Shorter, especially one-, two-, and three-line, inputs have seldom comments; larger inputs have approximately one comment per three to four lines of code. Sorting the above data `data` with respect to the number of lines of code yields the following distribution of the average number of comments as a function of the number of code lines of the inputs. Because the number of inputs with more than 20 lines of code is relatively small, the data have relatively large fluctuations on the right end of the graphic.

```
ListPlot[Select[{#[[1, 1]], (Plus @@ (Last /@ #))/Length[#]}& /@
                (Split[Sort[Reverse /@ Flatten[data, 1]],
                       #1[[1]] === #2[[1]]&]), First[#] < 50&]]
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

e) To count the number of successive square closing brackets, we read in all notebooks of the *Mathematica GuideBooks*, extract all input cells, delete all comments, and transform the inputs into a sequence of characters. After deleting whitespace, we use `Split` to separate groups of square closing brackets.

```
notebooks = Flatten[
     {Function[{c, n}, (c <> ToString[#] <> ".nb")& /@ Range[n]] @@@
      {{"1_Programming_", 6}, {"2_Graphics_", 3},
       {"3_Numerics_", 2}, {"4_Symbolics_", 3}},
      "Preface.nb", "0_Introduction.nb", "Appendix_A.nb"}];
```

```
fileNames = ToFileName[ReplacePart[
            "FileName" /. NotebookInformation[EvaluationNotebook[]],
             #, 2]]& /@ notebooks;

data = Table[
nb = Get[fileNames[[k]]];
(* the input cells *)
inputCells = Cases[nb, Cell[__, "Input", ___], Infinity];
(* the input strings *)
inputStrings = Which[Head[#[[1]]] === String, #[[1]],
                     Head[#[[1]]] === TextData,
                     StringJoin[Cases[#[[1, 1]], _String]],
                     True, Sequence @@ {}]& /@ inputCells;
(* the characters of the input strings *)
characters = DeleteCases[Characters[#],
                 (* ignore spaces and newlines *)
                   "\t" | "\n" | " "]& /@ inputStrings;
(* count sequences of "]" *)
{StringJoin[First[#]], Length[#]}& /@
   Split[Sort[Flatten[Cases[Split[#], {"]", ___}]& /@
             characters, 1]]], {k, 14}];
```

We add all results from the 14 chapters of the four volumes of the *GuideBooks*.

```
res = Sort[Flatten[data, 1],
        StringLength[#1[[1]]] <= StringLength[#2[[1]]]&] //.
         {a___, {α_, n_}, {α_, m_}, b___} :> {a, {α, n + m}, b};

res // TableForm
```

To a good approximation, we find that the probability $p_n^{(])}$ of n successive closing square brackets obeys $p_n^{(])} \sim \exp(-n)$.

```
logProbPlot[res_] :=
Module[{n = Plus @@ (Last /@ res)},
ListPlot[{#[[1]], Log[10, #[[2]]]}& /@
                   N[{StringLength[#[[1]]], #[[2]]/n}& /@ res],
         PlotJoined -> True, Axes -> False,
         Frame -> True, PlotRange -> All]]

logProbPlot[res]
```

Now, let us deal with all input written in `FullForm`. To obtain the `FullForm` version of the inputs, we have to interpret the inputs using `ToHeldExpression`. We then transform the resulting expressions into strings, strip out the enclosing `Hold[]` characters, delete whitespace, and proceed as above.

```
(* suppress messages *)
Off[Syntax::com]; Off[SyntaxQ::string]; Off[Trace::shdw];
Off[List::string]; Off[StringJoin::string]; Off[Precision::precsm];
```

```
data = Table[
nb = Get[fileNames[[k]]];
(* the input cells *)
inputCells = Cases[nb, Cell[__, "Input", ___], Infinity];
(* the interpreted inputs *)
heldInputs = If[SyntaxQ[#],
ToHeldExpression[#], Sequence @@ {}]& /@
 DeleteCases[Which[Head[#[[1]]] === String, #[[1]],
                   Head[#[[1]]] === TextData,
                   StringJoin[#[[1, 1]] /. StyleBox[s_, ___] :> s]]& /@
                                                  inputCells, Null, {1}];
(* the input strings *)
inputStrings = If[StringLength[#] > 6,
      StringDrop[StringDrop[#, -1], 5]]& /@
        (ToString[FullForm[#]]& /@ heldInputs);
(* the characters of the input strings *)
characters = DeleteCases[Characters[#],
                         (* ignore spaces and newlines *)
                           "\t" | "\n" | " "]& /@ inputStrings;
(* count sequences of "]" *)
{StringJoin[First[#]], Length[#]}& /@
   Split[Sort[Flatten[Cases[Split[#], {"]", ___}]& /@
                characters, 1]]], {k, 14}];
```

Using the `FullForm` versions of the inputs yields a different distribution. No `]]` for part extraction occur anymore, but many `Map`, `Apply`, …, that are written in their infix form contribute now with closing square brackets.

```
res = Sort[Flatten[data, 1],
           StringLength[#1[[1]]] <= StringLength[#2[[1]]]&] //.
      {a___, {α_, n_}, {α_, m_}, b___} :> {a, {α, n + m}, b};

res // TableForm
```

Again, we find that the probability $\tilde{p}_n^{(])}$ of n successive closing square brackets obeys approximatively $\tilde{p}_n^{(])} \sim \exp(-n)$.

```
logProbPlot[res]
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

f) We start by creating a list of all notebooks to be checked.

```
notebooks = (* Programming volume only *)
     ("1_Programming_"<> ToString[#] <> ".nb")& /@ Range[6];

fileNames = ToFileName[ReplacePart[
           "FileName" /. NotebookInformation[EvaluationNotebook[]],
            #, 2]]& /@ notebooks;

nbs = Get /@ fileNames;
```

We extract all cells containing *Mathematica* inputs (ignoring inline cells).

```
allCells = Flatten[#[[1]] //.
           Cell[CellGroupData[l_List, ___], ___] :> l]& /@ nbs;

inputCells = Cases[#, Cell[_?(FreeQ[#, _BoxData, {0, Infinity}]&),
                           "Input" | "Program", ___],
                   Infinity]& /@ allCells;
```

These is the number of input cells to be checked.

```
Length /@ inputCells
```

Given a cell of type `"Input"` or `"Program"`, the function `makeInputString` generates a single string of the actual input. Comments are stripped out and a single whitespace is prepended and appended. Newline characters are treated as a single empty space.

```
makeInputString[c:Cell[s_, "Input" | "Program", ___]] :=
StringReplace[StringJoin[" ",
Which[Head[s] === String, s,
      (* delete comments; concatenate pieces *)
      Head[s] === TextData, StringJoin[s[[1]] /. _StyleBox :> " "],
      True, Print[k]; CellPrint[c]], " "], {"\n" -> " ", "\t" -> " "}]
```

The function `spacingCorrectQ` tests if the string *s* has for all elements of `allowedNeighbors` "allowed" neighbors. `allowedNeighborsQ[`*s*`,` *characters*`,` *potentialLeftNeighbors*`,` *potentialRightNeighbors*`]` returns `True` if the character sequence *characters* inside the string *s* has a left neighboring character from the list *potentialLeft*-*Neighbors* and a right neighboring character from the list *potentialRightNeighbors*. `Any` indicates that any character can appear. So, for example, to the left of a semicolon `';'` any character can appear, but to the right an empty space or a closing bracket or a closing parentheses is allowed.

```
spacingCorrectQ[s_String] :=
With[{a = allowedNeighborsQ,
      (* special treatment of Increment and Decrement *)
      s = StringReplace[s, {"++" -> " +", "--" -> " -"}]},
 a[s, ";",   Any, {" ", "]", ")"}] &&
 a[s, ",",   Any, {" "}] &&
 a[s, "+",   {" ", "+", "(", "^", "[", "{"}, Any] &&
 a[s, "-",   {" ", "-", "(", "^", "[", "{", "`"}, Any] &&
 a[s, "=",   {" ", "=", ":", "^", "!", ">", "<"}, {" ", "=", "!", "."}] &&
 a[s, ":=",  {" ", "^"}, {" "}] &&
 a[s, "==",  {" ", "="}, {" ", "="}] &&
 a[s, "<",   {" ", "<"}, {" ", "=", "<", ">"}] &&
 a[s, ">",   {" ", "-", ":", ">", "<"}, {" ", "=", ">"}] &&
 a[s, "<>",  {" "}, {" "}] &&
 a[s, "===", {" "}, {" "}] &&
 a[s, "=!=", {" "}, {" "}] &&
 a[s, "->",  {" "}, {" "}] &&
 a[s, ":>",  {" "}, {" "}] &&
 a[s, "/.",  {" ", "/"}, {" "}] &&
 a[s, "//.", {" "}, {" "}] &&
 a[s, "//",  {" "}, {" ", ".", "@"}] &&
 a[s, "/;",  {" "}, {" "}] &&
 a[s, "@",   {" ", "/", "@"}, {" ", "@"}] &&
 a[s, "/@",  {" ", "/"}, {" ", "@"}] &&
 a[s, "@@",  {" ", "@"}, {" ", "@"}] &&
 a[s, "@@@", {" "}, {" "}] &&
 a[s, "&&",  {" "}, {" "}] &&
 a[s, "||",  {" "}, {" "}] &&
 a[s, "|",   {" ", "|"}, {" ", "|"}]]
```

The function `allowedNeighbors` finally locates the position of the character sequence of interest and checks their neighboring characters. We do not want to reimplement the *Mathematica* parser and we do not have to for our restricted purpose. Simply checking the left and right neighbors is enough for our purposes. A more refined treatment would take into account if the characters appear inside a string, for instance.

```
allowedNeighborsQ[s_String, characters_String,
                  potentialLeftNeighbors_, potentialRightNeighbors_] :=
Module[{posis = StringPosition[s, characters]},
 If[posis === {}, True,
     (* actual left neighbor characters *)
     leftCharacters = Union[StringTake[s, {#, #}]& /@
                               ((First /@ posis) - 1)];
     (* actual right neighbor characters *)
     rightCharacters = Union[StringTake[s, {#, #}]& /@
                               ((Last /@ posis) + 1)];
     (* are actual left neighbor characters allowed? *)
     If[potentialLeftNeighbors === Any, True,
        Complement[leftCharacters,
                   Append[potentialLeftNeighbors, "\""]] === {}] &&
     (* are actual right neighbor characters allowed? *)
     If[potentialRightNeighbors === Any, True,
        Complement[rightCharacters,
                   Append[potentialRightNeighbors, "\""]] === {}]]]
```

Here are two simple examples of the use of `allowedNeighborsQ`. The second input does not have a space after the semicolon.

```
allowedNeighborsQ["1 + 1; 2", ";", Any, {" ", "]", ")"}]

allowedNeighborsQ["1 + 1;2", ";", Any, {" ", "]", ")"}]
```

`spacingCorrectQ` tests for the neighbors of 25 character sequences at once. The second element of the following list does not have a space after one comma and no spaces around `->`.

```
{spacingCorrectQ["Plot[Sin[x], {x, 0, 1}, Frame -> True]}]"],
 (* the next input has spacing mistakes *)
 spacingCorrectQ["Plot[Sin[x],{x, 0, 1}, Frame->True]}]"]}
```

The six chapters of this book have about 5400 *Mathematica* input containing cells.

```
Length[Flatten[inputCells]]
```

Now, we check all of them. We observe some violations of our declared spacing rules. But reading the text surrounding these cells, we recognize that these violations were all intentional.

```
Do[(* make one input string *)
   input = makeInputString @ inputCells[[j, k]];
   (* check spacing and potentially print the problem *)
   If[Not[spacingCorrectQ[input]],
      CellPrint[Cell["∘ In Chapter " <> ToString[j] <> ":", "PrintText"]];
      CellPrint[Append[inputCells[[j, k]],
                C[Evaluatable -> False, FontColor -> GrayLevel[0.5]]] /.
                C -> Sequence]],
  {j, Length[inputCells]}, {k, Length[inputCells[[j]]]}];
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

g) We first implement some functions that extract the cells containing from a notebook. Then we extract the texts from these cells, split these texts into sentences, and finally into pairs of consecutive words.

```
(* extract cells containing text from a notebook *)
extractCells[nb_] := Cases[nb, Cell[_,
          "Text" | "TextDescription" | "ItemizedNoteBox", ___], Infinity];

(* if free of typesetting, convert styled text into plain text *)
toText[c_] := c /. StyleBox[s_String?LetterQ, __] :> s
```

```
(* extract cells containing text from a notebook *)
makeTexts[cells_] :=
Which[Head[#] === String, #,
          (* join pieces to one string *)
          Head[#] === List, StringJoin[#],
          True, Sequence @@ {}]& /@
            (Which[Head[#[[1]]] === String, #[[1]],
                         (* delete remaining box structures *)
                         Head[#[[1]]] === TextData,
                          DeleteCases[toText[#[[1, 1]]],
                               _CounterBox | _StyleBox | _ButtonBox |
                               _Cell, Infinity], True, Sequence @@ {}]& /@ cells);
```

```
(* split a given text into pieces *)
sentencePieces[text_String] :=
Module[{s, posis, seqs, fragments1, fragments2},
   s = StringJoin[StringReplace[text,
       {"[" -> " ",  "]" -> " ", "“" -> "", "”" -> "", "-" -> " ",
        "—" -> " ", "{" -> "", "}" -> "", "■" -> ""}], " "];
 (* are delimiters present *)
 If[posis = StringPosition[s, {".", "?", "!", ";", ":", "(", ")"}];
    posis =!= {}, λ = StringLength[s];
    (* make pieces *) seqs = ({-1, +1} + #& /@ posis);
    fragments1 = StringTake[s, #]& /@
              Join[{{1, seqs[[1, 1]]}}, {#[[1, 2]], #[[2, 1]]}& /@
                   Partition[seqs, 2, 1], {{seqs[[-1, 2]], λ}}],
    fragments1 = {s}];
 fragments2 = StringReplace[#, {"." -> "", "," -> "",
             "(" -> "", ")" -> "", ":" -> ""}]& /@ fragments1;
 DeleteCases[fragments2, "" | " " | "  "]]
```

```
(* split a sentence into a list of words *)
toWords[sentence_String] :=
Module[{s, λ, posis, words},
 (* use lower case words only *)
 s = FixedPoint[StringReplace[#, "  " -> " "]&, ToLowerCase[sentence]];
 λ = StringLength[s];
 (* find word delimiter " " *)
 posis = Partition[Flatten[{1, {-1, 1} + #& /@
                            StringPosition[s, " "], λ}], 2];
 (* return list of consecutive words *)
 words = StringTake[s, #]& /@ Map[Min[#, λ]&, posis, {-1}];
 Select[DeleteCases[words, ""], LetterQ]]
```

The function `makeNeighbors` forms the neighbors of all words of a sentence or a sentence fragment.

```
(* form neighbor pairs from a list of words *)
makeNeighbors[s_String] := Partition[toWords[s], 2, 1]
```

The function `spellCheck` returns the words from the list *words* that are not proper English words.

```
(* spell check a list of words *)
spellCheck[words_] :=
With[{(* an invisible notebook *)
      nb = NotebookPut[Notebook[{Cell[ToString[words], "Text"]},
                                 Visible -> False]], l = $ParentLink},
     LinkWrite[l, NotebookGetMisspellingsPacket[nb]];
     (NotebookClose[nb, Interactive -> False]; #)&[LinkRead[l]]]
```

These are the 17 files of the *GuideBooks* that we will use as the source for text.

```
notebooks =
     {"1_Programming_1.nb", "1_Programming_2.nb", "1_Programming_3.nb",
      "1_Programming_4.nb", "1_Programming_5.nb", "1_Programming_6.nb",
      "2_Graphics_1.nb", "2_Graphics_2.nb", "2_Graphics_3.nb",
      "3_Numerics_1.nb", "3_Numerics_2.nb",
      "4_Symbolics_1.nb", "4_Symbolics_2.nb", "4_Symbolics_3.nb",
      "Preface.nb", "0_Introduction.nb", "Appendix_A.nb"};
```

Using the above functions, we extract the texts from the notebooks and form all pairs of consecutive words.

```
data =
Table[nb = Get[ToFileName[ReplacePart[
               "FileName" /. NotebookInformation[EvaluationNotebook[]],
                                                  notebooks[[j]], 2]]];
      (* extract cells containing text *)
      cells = extractCells[nb];
      (* extract text *)
      texts = makeTexts[cells];
      (* extract sentences *)
      allSentencePieces =
      Flatten[Table[Check[sentencePieces[texts[[k]]],
                    (* to see potential problems *) print[k]],
              {k, Length[texts]}]];
      (* list of neighboring words *)
      Flatten[makeNeighbors /@ allSentencePieces, 1],
      {j, 1, Length[notebooks]}];

allPairs = Flatten[data, 1];
```

Because we often in the *GuideBooks* use descriptive multi-word-symbols for user-supplied variables that are not proper English words, we eliminate all pairs that contain such multi-word-symbols. After doing this we have about 445000 pairs of words.

```
badWords = (spellCheck @
             Complement[Union[Flatten[allPairs]], ToLowerCase /@ Names["*"]]

(* non-English words -> 0 *)
dRules = Dispatch[(# :> 0)& /@ badWords];
finalPairs = Cases[DeleteCases[allPairs /. dRules,
                               badWords, {-1}], {_String, _String}];
Length[finalPairs]
```

Now, we eliminate doubles and count the number of different pairs—more than 100000 different pairs occur.

```
wordsAndNumberNumbers = Sort[{Length[#], #[[1, 1]]}& /@
           (Union /@ Split[Sort[finalPairs], #1[[1]] === #2[[1]]&])];

Λ = Plus @@ (First /@ wordsAndNumberNumbers)
```

Here are the words with the most potential neighbors and the number of different neighbors.

```
Take[wordsAndNumberNumbers, -50] // Reverse
```

On average, the words of the *GuideBooks* have about 14 different neighbors. This is not much, but for a computer-system-related book from a nonnative author, one does not expect the word variety of a novel.

```
N[Plus @@ (First /@ #)/Length[#]]&[wordsAndNumberNumbers]
```

The next graphic shows a logarithmic plot of the data from `wordsAndNumberNumbers`.

```
logData = Log[10, N[First /@ wordsAndNumberNumbers]];
ListPlot[logData, PlotRange -> All]
```

Next, we bin the data `logData`.

```
makeBins[l_, δ_] := {First[#] δ, Length[#]}& /@ Split[Round[Sort[l]/δ]]
d = {First[#], Log[10, #[[2]]/Λ]}& /@ makeBins[logData, 0.26];
```

For a guide for the eye we calculate two best-fit curves for the data. The functions `Fit` is used here, we will discuss it in Chapter 1 of the Numerics volume [302✱].

```
χ = 1.7;
y1[x_] = Fit[Select[d, #[[1]] <= χ&], {1, x}, x];
y2[x_] = Fit[Select[d, #[[1]] >= χ&], {1, x}, x];
```

So, we finally arrive at the following graphic. While we analyzed a comparatively small amount of data, the typical two power law structure is clearly visible. The transition point is around 50 neighbors.

```
ListPlot[d, PlotRange -> All, Frame -> True, Axes -> False,
          PlotStyle -> {GrayLevel[0], PointSize[0.025]},
          Prolog -> {Hue[0], Thickness[0.01],
            Line[{{0, y1[0]}, {χ, y1[χ]}}],
            Line[{{χ, y2[χ]}, {d[[-1, 1]], y2[d[[-1, 1]]]}}]}]
```

To obtain a more reliable value for the crossover point than from visual inspection, we plot the residue of fits with two straight lines as a function of the crossover point. The following graphic shows the resulting residue, the curves of different color represent different bin sizes.

```
δV[binSize_, χ_?NumberQ] :=
Module[{d, d1, d2, y1, y2},
 (* log bin data *)
 d = {First[#], Log[10, #[[2]]/Λ]}& /@ makeBins[logData, binSize];
 (* the two linear fits *)
 y1[x_] = Fit[d1 = Select[d, #[[1]] <= χ&], {1, x}, x];
 y2[x_] = Fit[d2 = Select[d, #[[1]] >= χ&], {1, x}, x];
 (* residue *)
 ((Plus @@ (Abs[y1[#1] - #2]& @@@ d1)) +
  (Plus @@ (Abs[y2[#1] - #2]& @@@ d2)))]

Plot[Evaluate[Table[δV[binSize, ξ], {binSize, 0.2, 0.4, 0.2/25}]],
     {ξ, 1, 2}, PlotPoints -> 20,
     PlotStyle -> Table[Hue[x], {x, 0, 0.8, 0.8/25}]]
```

The average value of the minima is about 1.7.

```
Module[{lines = Cases[%, _Line, Infinity], lineData, min},
  Sum[lineData = lines[[k, 1]];
      min = Min[Last /@ lineData];
      (#[[-1, 1]] + #[[1, 1]])/2&[Cases[lineData, {_, min}]],
      {k, Length[lines]}]/Length[lines]] // N[#, 3]&
```

For a similar distribution for references, see [123✱].

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

5. Tube Points

Here the lists are manipulated for $n = 8$ and $n = 5$.

```
n = 8; m = 5;

S[l__] := StringJoin[ToString /@ {l}]

points = Table[{S[p, i, x], S[p, i, y], S[p, i, z]}, {i, n}]
```

```
radii = Table[StringJoin["r", ToString[i]], {i, n}]

radii = Table[𝒮[r, i], {i, n}]

vecv = Table[{𝒮[v, i, x], 𝒮[v, i, y], 𝒮[v, i, z]}, {i, n}]

vecu = Table[{𝒮[u, i, x], 𝒮[u, i, y], 𝒮[u, i, z]}, {i, n}]

{cos = Table[𝒮[c, i], {i, m}], sin = Table[𝒮[s, i], {i, m}]}
```

Here is the "obvious" implementation using `Table`.

```
version1 = Table[Expand[points[[i]] + radii[[i]] (cos[[j]] vecv[[i]] +
                                                 sin[[j]] vecu[[i]])],
                 {i, n}, {j, m}];

Short[version1, 14]
```

Next, we give a somewhat more elegant and faster formulation. Its operation will become obvious after some thought.

```
version2 = MapThread[
     Map[Function[x, #1 + x], #2]&,
         {points, Partition[Apply[Plus,
                Distribute[{radii Transpose[{vecv, vecu}],
                            Transpose[{cos, sin}]},
                           List, List, List, Times],
             {1}], m]}];
```

Here is another version.

```
version3 = Transpose[points + #& /@
                (Outer[Times, cos, radii vecv] +
                 Outer[Times, sin, radii vecu]), {2, 1, 3}];
```

The three results are equal.

```
version1 == version2 == version3
```

Here is a comparison of the needed computational times.

```
Timing[Do[Table[Expand[
             points[[i]] + radii[[i]] (cos[[j]] vecv[[i]] +
                                       sin[[j]] vecu[[i]])],
       {i, n}, {j, m}], {100}]]

Timing[Do[MapThread[
     Map[Function[x, #1 + x], #2]&,
         {points, Partition[
           Apply[Plus,
                Distribute[{radii Transpose[{vecv, vecu}],
                            Transpose[{cos, sin}]},
                           List, List, List, Times],
             {1}], m]}], {100}]]

Timing[Do[Transpose[points + #& /@
               (Outer[Times, cos, radii vecv] +
                Outer[Times, sin, radii vecu]), {2, 1, 3}], {100}]]
```

The result is not surprising because, in the first version, all lists have to be manipulated repeatedly to extract the needed parts, whereas in the second and third version, the lists are always treated at once.

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

6. All Subsets

We look at what happens step by step.

1) `Union` removes all elements from the list `l` that appear more than once.

2) `{{}, {#}}& /@ …` makes a list with the elements `{{}, {`*e*`}}` for every element *e* in the list `l`.

3) `Distribute[…, List, List, List, Union]` does the actual work. It is based on "multiplying out" `{{}, {`e_1`}} × {{}, {`e_2`}} × {{}, {`e_3`}} ×⋯× {{}, {`e_n`}}`.

We now look at the result with another (`union` instead of `Union`) fifth argument of `Distribute` to see what happens.

```
Distribute[{{{}, {a}}, {{}, {b}}, {{}, {c}}}, List, List, List, union]
```

4) The `Union` in the fifth argument of `Distribute` removes the superfluous empty lists and combines elements that belong together in a set. Here is `allSubsets` in action.

```
allSubsets[l_List] := Sort[Distribute[{{}, {#}}& /@
                                Union[l], List, List, List, Union]]
```

```
allSubsets[{a, b, c, d}]
```

The last result is identical to the one returned from the built-in function `Subsets`.

```
Subsets[{a, b, c, d}]
```

Now let us deal with the sum multidimensional sum $\mathcal{A}(k_1, k_2, \ldots, k_n)$. Here is a direct implementation of $\mathcal{A}(k_1, k_2, \ldots, k_n)$.

```
𝒜[hL_] := With[{K = Times @@ hL},
               1/K Sum[Times @@ Floor[hL j/K], {j, 0, K - 1}]]
```

We can speed up `𝒜[`*hL*`]` by forming the product in the body of the sum only once.

```
𝒜S[hL_] := With[{K = Times @@ hL},
               1/K Sum[Evaluate[Times @@ Floor[hL j/K]],
                       {j, 0, K - 1}]]
```

Using a slight adaptation of the last `Distribute[...]`, it is straightforward to implement the following one-liner for calculating $\mathcal{A}(k_1, k_2, \ldots, k_n)$.

```
𝒜C[l_] := Times @@ (l - 1) + Plus @@
    ((-1)^#1 Sum[j/#2 Times @@ Floor[j #3/#2],
                 {j, 0, #2 - 1}]&[
     Length[#], GCD @@ l[[#]], Complement[l, l[[#]]]]& /@
                          Rest[Subsets[Range[Length[l]]]])
```

Next, we use the three implementations with the first five primes.

```
{𝒜[#] // Timing, 𝒜S[#] // Timing, 𝒜C[#] // Timing}&[
                                    {2, 3, 5, 7, 11, 13}]
```

Calculating $\mathcal{A}(p_1, p_2, \ldots, p_{10})$ directly would require summing about 6.5 10^9 terms. The subset summation ranges over 1023 subsets and all together 5120 floor terms only.

```
𝒜C[{2, 3, 5, 7, 11, 13, 17, 19, 23, 29}]
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

7. Moessner's Process, Ducci's Iterations, Matrix Product

a) First, here is a possible implementation.

```
strikeList[ord_Integer?Positive, num_Integer?Positive] :=
Fold[Rest[FoldList[Plus, 0,
                Delete[#1, List /@ Range[#2, Length[#1], #2]]]]&,
        Range[num ord], Range[ord, 2, -1]]
```

This formulation is relatively efficient. `Range[num ord]` produces the initial list of numbers. `List /@ Range[#2, Length[#1], #2]` creates a list with the numbers to be eliminated. `Delete[…]` removes these elements, and `FoldList[Plus, …]` sums the resulting numerical sequences. `Rest` is needed to get rid of the 0 at the beginning of the summation. `Fold` takes care of the work of removing every *i*th element, …, every second element. To be able to follow the removal process somewhat better, we replace `Fold` by `FoldList`.

```
strikeListLong[ord_Integer?Positive, num_Integer?Positive] :=
FoldList[Rest[FoldList[Plus, 0,
                Delete[#1, List /@ Range[#2, Length[#1], #2]]]]&,
         Range[num ord], Range[ord, 2, -1]]
```

Here is an example.

```
strikeListLong[4, 4]
```

Using `Trace`, we see in detail how the program `strikeList` works.

```
Trace[strikeList[3, 2]]
```

We now run `strikeList` for `ord` = 2, 3, 4, and 5.

```
strikeList[2, 12]

strikeList[3, 12]

strikeList[4, 12]

strikeList[5, 12]
```

Here is a comparison of the last results with the first 12 fifth powers.

```
Range[12]^5
```

This result indicates that the resulting lists for $n = 4$ and 5 are also powers for small n. Actually, not only for small n, but for all n. For an explanation, see [218✱], [232✱], [142✱], and [183✱].

Note that there are other, similar identities. For instance, the nth-order differences of the sequence $1^n, 2^n, \ldots$ is just $n!$ [72✱].

```
SchubertRelation[ord_Integer?Positive, len_Integer?Positive] :=
    (-1)^ord Nest[Apply[Subtract, Partition[#, 2, 1], {1}]&,
                  Array[#^ord&, len], ord] ==
                  Array[Evaluate[ord!]&, len - ord] /; len >= ord

Table[SchubertRelation[i, j], {i, 48}, {j, i, 48}] // Flatten // Union
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

b) Using `FixedPointList`, this construction is easily implemented. Here is an example.

```
FixedPointList[Abs[Apply[Subtract, (* make pairs *)
     Partition[Append[#, First[#]], 2, 1], {1}]]&, {41, 71, 81, 13}]
```

Interestingly, this process ends in four equal numbers. Let us check 1512 more examples.

```
DucciChain[l:{_Integer?Positive..}] :=
Drop[FixedPointList[Abs[Apply[Subtract,
      Partition[Append[#, First[#]], 2, 1], {1}]]&, l], -2]
```

Here is an example.

```
DucciChain[{111, 112, 113, 114}]

Union[Flatten[Table[Equal @@ Last[DucciChain[{a, b, c, d}]],
                    {a, 30, 35}, {b, 67, 72}, {c, 56, 62}, {d, 89, 94}]]]
```

Here is a visualization of the convergence process. (We discuss the command `Random` in the next chapter.) The first two numbers and the second two numbers of the four-element list are used to form Cartesian coordinates. The solid lines connect points from one iteration stage, and the dotted lines show the iteration step.

```
Show[GraphicsArray[#]]& /@
  Partition[Table[(* make the table of 4 x 4 pictures *)
    Graphics[{(* make lines in both directions *)
             Thickness[0.001], Line /@ #,
            {Dashing[{0.03, 0.03}], Thickness[0.001],
             Line /@ Transpose[#]},
            {PointSize[0.02], Point[Last[#][[2]]]}}&[
             Map[Partition[#, 2]&, DucciChain[
                 (* four randomly chosen start integers *)
                 Table[Random[Integer, {1, 100}], {4}]]]],
            Frame -> True, FrameTicks -> None,
            AspectRatio -> 1, PlotRange -> All], {16}], 4]
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

c) It is straightforward to implement the matrix product. We form the product as long as the result deviates from e by more than 10^{-1000}.

```
e = N[E, 1000];
A = IdentityMatrix[2];
k = 1;
While[Abs[(A[[1, 1]] + A[[2, 1]])/
          (A[[1, 2]] + A[[2, 2]]) - e] > 10^-1000,
      A = {{2k, 2k - 1}, {2k - 1, 2k - 2}}.A; k++];
```

After 203 steps, we obtain 1000 correct digits. At this point, the matrix has integer elements with 499 digits.

```
{k, N[A]}
```

The ratios of elements of the matrix allow to give lower and upper bounds for e.

```
$MaxExtraPrecision = 1000;
A[[2, 1]]/A[[2, 2]] < E < A[[1, 1]]/A[[1, 2]]
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

8. Triangles, Group Elements, Partitions, Stieltjes Iterations

a) First, we look at the result.

```
NestedTriangles[n_Integer?Positive] :=
(Function[{x, y}, x.#& /@ y] @@ #)& /@
   Distribute[{Table[{{ Cos[i Pi/2], Sin[i Pi/2]},
                      {-Sin[i Pi/2], Cos[i Pi/2]}}, {i, 0, 3}],
               Flatten[NestList[#/2&, {{{1, 1}, {3, +1}, {1, 3}},
                                      {{1, 0}, {2, -1}, {2, 1}}}, n], 1]},
              List];

Show[Graphics[Polygon /@ NestedTriangles[6]],
     AspectRatio -> Automatic, PlotRange -> All]
```

Here is how it works. The `{{1, 1}, {3, 1}, {1, 3}}, {{1, 0}, {2, -1}, {2, 1}}` are the coordinates of the vertices of two initial triangles. The part `Nest[#/2&, …]` produces n reduced in size and moved toward the origin {0, 0} copies of the triangle. `Flatten` removes the inner brackets so that only lists with coordinates remain. `Table[{{ Cos[i Pi/2], Sin[i Pi/2]}, {-Sin[i Pi/2], Cos[i Pi/2]}}, {i, 0, 3}]` creates four rotation matrices corresponding to rotation angles 0°, 90°, 180°, and 270°. `Distribute[{..., ...}, List]` forms all possible combinations of the triangles and rotation angles. Finally, the following function performs the rotation of all vertices of a triangle using a given rotation matrix: `Function[{x, y}, x.#& /@ y] @@ #)& /@ ….`

Σ (* session summary *) `` TMGBs`PrintSessionSummary[] ``

b) Let us run the code to see what happens.

```
FixedPoint[Union[Flatten[Outer[Function[C, #]& @
                 Simplify[#1[#2[C]]]&, #, #]]]&,
           {Function[C, -C], Function[C, (C + I)/(C - I)]}]
```

We start with two pure functions, and new pure functions are formed by composition with the inner argument `C`. After the composition has been done, the result is simplified and transformed again into a pure function. This evaluation happens by applying `Outer` with every possible combination of pure functions, until no new ones are generated. This procedure only makes sense when the functions form a group under composition, so that this process finishes naturally at some stage. In the example above, the group under consideration is the tetrahedral group.

Σ (* session summary *) `` TMGBs`PrintSessionSummary[] ``

c) The function `PartitionsLists` generates a list of all weakly decreasing sequences of nonnegative numbers summing to n. Let us discuss what is done inside `partitionsLists`. First, a list of the form `{{`n`, 0, 0,… 0}}` with one sublist with $n-1$ zeros is created. Then the function `Complement[...]&` is repeatedly applied to these sublists until the result no longer changes. At each step, new sublists are formed from each sublist by moving a “unit” to the right in such a way that we form a new weakly decreasing sequence. The two rules form such sequences if possible, `Union` eliminates doubles, and the function `ReplaceList` makes sure that we generate all possible ones. Then, using `Complement`, the sequences that were already present are eliminated. The results returned contain all newly created sublists at each step.

```
PartitionsLists[n_Integer?Positive] := Drop[FixedPointList[
 Complement[Union[Flatten[ReplaceList[#,
  {{a___, b_, c_, d___} :> {a, b - 1, c + 1, d} /; b - c >= 2,
  {a___, b_, c:(d_ ...), e_, f___} :> {a, b - 1, c, e + 1, f} /;
              b - 1 == d == e + 1}]& /@ #, 1]], #]&,
                           {{n, ##}& @@ Table[0, {n - 1}]}], -2]
```

Inspecting the following input demonstrates how `partitionsLists` works.

```
PartitionsLists[4]
```

Here is a slightly larger example. For brevity, we display only the length of the sublists.

```
Length /@ PartitionsLists[33]
```

The built-in function `PartitionsP[n]` returns the number of weakly decreasing sequences of nonnegative numbers that sum to n. This shows that the last input generated all possible of the more than 10000 sequences.

```
{Plus @@ %, PartitionsP[33]}
```

The following graphic shows how many new sequences were created at each step [108✱].

```
ListPlot[%%]
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

d) First, the protected symbol `Table` is unprotected. Then an option `Heads` is added to `Table`. The option setting of the `Heads` option is the head to be used instead of `List` of the resulting nested expression. First, the `Table` command without the `Heads` option is evaluated and then the `List` heads are replaced with the given heads. In case the number of given heads is less than the depth of the nested list generated by `Table` the heads are used cyclically.

```
Unprotect[Table];

Table[body_, iters__, Heads -> l_List] :=
With[{d = Length[{iters}]},
Fold[Apply[First[#2], #1, {Last[#2]}]&, Table[body, iters],
        Reverse[MapIndexed[{#1, #2[[1]] - 1}&,
        Take[Flatten[Table[l, {d}]], d]]]]]

Table[body_, iters__, Heads -> l_] :=  Table[body, iters, Heads -> {l}]
```

Here are some examples.

```
Table[Subscript[a, i, j], {i, 3}, {j, 3}, {k, 3}, Heads -> {A, B, C}]

Table[Subscript[a, i, j], {i, 2}, {j, 3}, {k, 4}, Heads -> {𝒜, ℬ}]
```

If only one head specification is supplied, it does not have to be enclosed in a list.

```
Table[Subscript[a, i, j], {i, 2}, {j, 2}, Heads -> 𝒜]
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

e) Let us begin analyzing 𝕡. 𝕡 is a list of lists of all ordered n-tuples ($n = 1, \ldots, \lambda$) of the integers $1, 2, \ldots, \lambda$. It is generated by recursively adding larger integers to the lists of already existing ones. Here this is demonstrated.

```
𝕡F[λ_] := NestList[Flatten[
      Outer[Join, {#}, List /@ Range[Last[#] + 1, λ], 1]& /@  #, 2]&,
                 List /@ Range[λ], λ - 1]

𝕡F[3]

𝕡F[5]
```

The `FixedPointList[…]` in $\mathcal{SA}$ starts with the list l and iterates the map $(l = \{l_1, l_2, \ldots, l_\lambda\}) \to \{\mu_\lambda(l), \mu_{\lambda-1}(l), \ldots, \mu_1(l)\}$. Here $\mu_k(l) = {}^{(k)}l / {}^{(k-1)}l$ and ${}^{(k)}l$ is the arithmetic mean of all products of k different numbers of the list l (we assume ${}^{(0)}l = 1$). The `Apply[Times, ...]` forms the products, `Apply[(Plus[##])/Length[{##}]&, …]` forms the arithmetic means, and `Divide @@@ Partition[…]` forms the quotients. Here one iteration step is shown for a symbolic list 𝔩 with five elements.

```
fplStep[𝕡_] := Function[ℓ,
      Divide @@@ Partition[Append[Reverse[Apply[(Plus[##])/Length[{##}]&,
        Apply[Times, Map[ℓ[[##]]&, 𝕡, {-2}], {2}], {1}]], 1], 2, 1]]

fplStep[𝕡F[5]][Array[Subscript[𝔩, #]&, {5}]]
```

The condition finally restricts the application of 𝒮𝒜 to lists containing only numeric elements, of which at least one must be approximate. This allows `FixedPointList` to terminate.

Now let us look at two examples of 𝒮𝒜 at work for two numeric lists.

```
𝒮𝒜[𝕝_List] :=  With[{λ = Length[𝕝]},
     Module[{𝕡 = NestList[Flatten[
      Outer[Join, {#}, List /@ Range[Last[#] + 1, λ], 1]& /@ #, 2]&,
                List /@ Range[λ], λ - 1]},
      FixedPointList[Function[ℓ,
      Divide @@@ Partition[Append[Reverse[Apply[Plus[##]/Length[{##}]&,
         Apply[Times, Map[ℓ[[##]]&, 𝕡, {-2}], {2}], {1}]], 1], 2, 1]], 𝕝]]]
         (Or @@ (InexactNumberQ /@ 𝕝)) && (And @@ (NumericQ /@ 𝕝))
```

```
(* use high-precision numbers *)
𝒮𝒜[N[{1, 2, 3}, 22]]
```

```
𝒮𝒜[N[{Pi, E, GoldenRatio, EulerGamma}]]
```

We see that the iterations converge and all elements of the list become equal. By observing that $\prod_{k=1}^{\lambda} \mu_k(l)$ does not change in the iterations, it is easy to show that the fixed point of these iterations is $\left(\prod_{k=1}^{\lambda} \mathbb{l}_k\right)^{1/\lambda}$ [288✱], [229✱].

```
{(1 2 3)^(1/3), (Pi E GoldenRatio EulerGamma)^(1/4)} // N
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

f) As the name of the function implies, `pseudoRandomTree` tries to build a random tree structure.

𝕣 defines a pseudorandom function that yields 0 with probability 1/2 and 2 with probability 1/2. The pseudorandom is based on the rounded functions value of a multiple of sin at far apart integers. The definition for 𝕥 is the main ingredient of the function `pseudoRandomTree`. The expression `Line[{`x`, `y`}, 𝕥[]]` first generates a pseudorandom integer through a call to 𝕣. When the integer is zero, the process stops. When the integer is 2, two `Line`-objects with a second argument are created and two further `Line`-objects of the form `Line[{`x'`, `y'`}, 𝕥[]]` are formed. The x-coordinate is increased by one and the x-coordinate, starting for each x from 0, is consecutively increased with each corresponding call to 𝕣 that gave 2 for the same x (this is done through the auxiliary function 𝓎). Starting from `Line[{0, 0}, 𝕥[]]`, we than let things loose. For most values of `kStart`, the recursive calls to 𝕥 will soon die (they die with probability one at some time). The result of this evaluation we call `tree`. `symmetrizeRules` extracts all `{`x`, `y`}` and symmetrizes them with respect to y so that for a given value of x, the y-values lie in the interval [$yMin(x)$, $yMax(x)$]. In the last step, the tree `tree` is symmetrized through the dispatched rule set `symmetrizeRules`, a `List` structure is added inside the `Line`-objects and a `Graphics`-object is formed and returned.

```
pseudoRandomTree[kStart_] :=
Module[{𝕣, k, 𝓎, 𝕥, tree, symmetrizeRules},
  (* pseudorandom function returning 0 or 2; mean == 1 *)
 (* singular "good choice":
   RealDigits[Pi, 18, 1000][[1]] and kStart = 0 *)
 𝕣 := If[IntegerPart[Abs[Sqrt[2] Sin[Pi k Sin[k = k + 1]]]] === 0,
        0, 2];
 (* initialize k and 𝓎 *)
 k = kStart; 𝓎[_] := -1;
 (* recursive definition for 𝕥;
  if 𝕣 yields true, make two new branches *)
 𝕥 /: Line[{x_, y_}, 𝕥[]] :=
         Table[{Line[{x, y}, {x + 1, 𝓎[x + 1] = 𝓎[x + 1] + 1}],
                Line[{x + 1, 𝓎[x + 1]}, 𝕥[]]}, {i, 𝕣}];
 (* form a tree *)
 tree = Line[{0, 0}, 𝕥[]];
 (* symmetrize tree with respect to y *)
 symmetrizeRules = Dispatch[Flatten [Function[l,
                    (# -> (# - {0, l[[-1, 2]]/2}))& /@ l] /@
                   Split[Union[DeleteCases[Level[tree, {-2}], {}]],
                         #1[[1]] === #2[[1]]&]]];
 (* return Graphics-object *)
 Graphics[(* form symmetrized tree *)
         tree /. symmetrizeRules /. Line[l__] :> Line[{l}],
         Frame -> True]]
```

Here are two examples. For *kStart* = 1, we get an empty tree; for *kStart* = 2, we get a nontrivial tree.

```
Table[pseudoRandomTree[k0] // InputForm, {k0, 2}]
```

Here this tree is shown.

```
Show[pseudoRandomTree[2]]
```

The next graphic shows the number of `Line`-objects in the resulting trees as a function of *kStart*.

```
ListPlot[Table[{k0, Count[pseudoRandomTree[k0], _Line, Infinity]},
               {k0, 1000}], PlotRange -> All]
```

We now show two larger trees. Because 𝕥 is potentially called many times recursively, we change the default value of `$RecursionLimit`.

```
$RecursionLimit = Infinity; $MaxExtraPrecision = 1000

Show[GraphicsArray[{pseudoRandomTree[836084275711],
                    pseudoRandomTree[506626351403]}]]
```

We end with a very big tree—it lives for many iterations and has nearly 100000 lines.

```
{(* depths and maximal width *)
 {Max[#1], 2Max[#2]}& @@
    Transpose[Level[Cases[#, _Line, Infinity], {-2}]],
 (* number of lines *)
 Count[#, _Line, Infinity]}&[pseudoRandomTree[914977508823]]
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

9. $\varepsilon\varepsilon \to \Sigma\delta\cdots\delta$, $\mathrm{Tr}(\gamma_{\mu_1}.\gamma_{\mu_2}.\cdots.\gamma_{\mu_{2n}})$, tanh Identity, Multidimensional Determinant

a) Here is one possible implementation. We do not give the explicit definition of $\varepsilon_{\nu\ldots\pi}$ and $\delta_{\nu\mu}$ here. We first program the case $r = n$. Several things have to be taken into consideration.

Because of the summation convention, the identity above holds only for variables that appear twice. It does not hold for numbers. But we are not sorting out the variables using `_Symbol`, but rather the numbers using `?(FreeQ[#, _Number]&)`, because a variable could be of type `a[2]` (i.e., it does not have head `Symbol`). Indexed variables will often apply in practical calculations when many indices exist and when they are “automatically” generated.

Because we want to find a rule for a product of Levi–Civita tensors, we have to input the rule via `TagSetDelayed`, which avoids the rule to be attached to `Times`, which would slow down `Times` considerably.

Because the variables appearing twice can be anywhere in the expression `LeviCivitaε`, whereas the Levi-Civita tensor has to be multiplied by -1 if two arguments are interchanged, we have to determine whether we have an even or an odd permutation. This is done in two steps:

- Changing from the given order of the arguments to the canonical normal form
- Changing variable order to the form with the variables appearing twice at the beginning.

The antisymmetrization is accomplished with `Permutations` along with the signature of the resulting permutations. For symmetry, we use `Kroneckerδ` instead of `KroneckerDelta`.

```
(* complete contraction, no tensor index remains *)
LeviCivitaε/: LeviCivitaε[var__?(FreeQ[#, _Number]&)] *
LeviCivitaε[var__?(FreeQ[#, _Number]&)] := Length[{var}]!;

(* the typical case *)
LeviCivitaε/: LeviCivitaε[var1__] LeviCivitaε[var2__] :=
Module[{commonIndices, rest1, rest2, s1, s2, ex, from},
(* the indices both have *)
commonIndices = Intersection @@
         (Select[#, Function[y, !NumberQ[y]]]& /@ {{var1}, {var2}});
(* the indices that exist only once *)
rest1 = Complement[{var1}, commonIndices];
rest2 = Complement[{var2}, commonIndices];
(* reordering indices and keep track of sign changes *)
s1 = Signature[{var1}]/Signature[Join[commonIndices, rest1]];
s2 = Signature[{var2}]/Signature[Join[commonIndices, rest2]];
(* the new indices pairs *)
ex = ({rest1, #, Signature[#]}& /@ Permutations[rest2])/Signature[rest2];
(* make Kronecker symbols *)
from = Plus @@ Apply[Times, {#[[3]],
                    Thread[Kroneckerδ[#[[1]], #[[2]]]]}& /@ ex, 2];
Length[commonIndices]! s1 s2 from]
```

We now try out the program for three dimensions.

```
LeviCivitaε[a, b, c] LeviCivitaε[a, b, c]

LeviCivitaε[a, b, c] LeviCivitaε[a, b, f]

LeviCivitaε[a, b, c] LeviCivitaε[a, e, f]

LeviCivitaε[a, b, c] LeviCivitaε[g, e, f]
```

Here is a short test for our function using the last result.

```
Table[Signature[{a, b, c}] Signature[{g, e, f}] -
       (% /. Kroneckerδ -> KroneckerDelta),
       {a, 0, 1}, {b, 0, 1}, {c, 0, 1},
       {g, 0, 1}, {e, 0, 1}, {f, 0, 1}] // Flatten // Union
```

Here is the product of two four-dimensional (4D) Levi-Civita tensors, written in a more traditional format.

```
LeviCivitaε[α, β, γ, ε] LeviCivitaε[ρ, μ, ν, σ] /.
                              Kroneckerδ[i__] -> Subscript[δ, i]
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

b) The function `sumTerms` calculates the antisymmetrized sum for a given *n*.

```
sumTerms[n_] := sumTerms[n] =
(Evaluate[signature[{##}] Product[A[j][k[j], Slot[j]], {j, n}]*
 KroneckerDelta[b, Slot[n + 1]]]& @@@
 Permutations[Append[#, a]& @ Table[k[j], {j, n}]]) /.
                                   signature -> Signature;
```

Here is an abbreviated form for $n = 2$. The antisymmetrized sum contains six terms.

```
(Plus @@ sumTerms[2]) /.
  KroneckerDelta[a_, b_] :> Subscript[δ, a, b] /.
  A[l_][k[i_], k[j_]] :>
     Subsuperscript[A[l], Subscript[k, i], Subscript[k, j]]
```

The number of summands is $(n+1)!$.

```
Table[{n, Length[sumTerms[n]]}, {n, 2, 6}]
```

The function 𝕤 sums over the doubly occurring indices for given *a* and *b*.

```
𝕤[a_, b_, n_] :=
Sum[(* sum over all terms from sumTerms[n] *)
    Sum[(* sum over the doubly occurring indices *)
        Evaluate[sumTerms[n][[i]]],
        Evaluate[Sequence @@ Table[{k[j], n}, {j, n}]]],
                                   {i, Length[sumTerms[n]]}];
```

Now, we carry out the summations for all *a* and *b*.

```
Table[𝕤[a, b, 2], {a, 2}, {j, 2}] // Timing

Table[𝕤[a, b, 3], {a, 3}, {j, 3}] // Timing

Table[𝕤[a, b, 4], {a, 4}, {j, 4}] // Timing
```

The case $n = 5$ is feasible, but using the function 𝕤 we would have to store large intermediate expressions. So, we carry out the sum term by term and merge the new terms with the old ones as soon as possible. The following input does this for $a = 1$, $b = 2$.

```
Block[{n = 5, a = 1, b = 2, sum = 0},
Do[sum = sum +
   Sum[(* sum over the doubly occurring indices *)
       Evaluate[sumTerms[n][[i]]],
       Evaluate[Sequence @@ Table[{k[j], n}, {j, n}]]],
       {i, Length[sumTerms[n]]}]; sum] // Timing
```

The remaining 24 pairs for $\{a, b\}$ can be treated in a similar way. So the case $n = 5$ is explicitly doable.

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

c) We will denote the Dirac matrices by `γ[μ]`. The function `DiracTrace` calculates the trace by calling the function diracTraceAux. For the function diracTraceAux, only the number of Dirac matrices matters; their indices are irrelevant.

```
DiracTrace[HoldPattern[Dot[Γ__γ]], η_] :=
Module[{indices = First /@ {Γ}, n = Length[{Γ}]/2},
 4 (diracTraceAux[n, η] /. (* use actual indices *)
         Apply[Rule, Transpose[{Range[2n], indices}], {1}])] /;
                                           EvenQ[Length[{Γ}]]

DiracTrace[HoldPattern[Dot[Γ__γ]], η_] := 0 /; OddQ[Length[{Γ}]]
```

The function diracTraceAux calculates the trace of $2n$ Dirac matrices. diracTraceAux takes into account only the minimal number of pairs by constructing such lists of pairs that obey the orderings $\mu_{i_1} < \mu_{i_3} < \cdots < \mu_{i_{2n-1}}$ and $\mu_{i_1} < \mu_{i_2}, \mu_{i_3} < \mu_{i_4}, \ldots, \mu_{i_{2n-1}} < \mu_{i_{2n}}$.

```
diracTraceAux[n_, η_] :=
Module[{l = Range[2n], firstSymbolsList, prePairs, lastSymbols, pairs},
(* the ordered list of first indices of the pairs *)
firstSymbolsList = Flatten[Table[Evaluate[Table[i[k], {k, n}]],
      Evaluate[Sequence @@
        Table[{i[k], If[k == 1, 1, i[k - 1] + 1], 2n}, {k, n}]]], n - 1];
(* potential pairs *)
prePairs = Flatten[(firstSymbols = #;
   lastSymbols = Complement[l, firstSymbols];
   Transpose[{firstSymbols, #}]& /@
             Permutations[lastSymbols])& /@ firstSymbolsList, 1];
(* check ordering within pairs *)
pairs = Select[prePairs, (And @@ Map[OrderedQ, #])&];
(* take into account signature and sum result *)
(Plus @@ ((Signature[Flatten[#]] Times @@ Apply[η, #, {1}])& /@ pairs))]
```

Here is an example of the output of diracTraceAux.

```
diracTraceAux[2, η]
```

Now let us calculate the traces of the actual products.

```
f1[μ_, ν_] = DiracTrace[γ[μ].γ[ν], η]

f2[μ_, ν_, ρ_, σ_] = DiracTrace[γ[μ].γ[ν].γ[ρ].γ[σ], η]

f3[μ_, ν_, ρ_, σ_, τ_, ξ_] = DiracTrace[γ[μ].γ[ν].γ[ρ].γ[σ].γ[τ].γ[ξ], η]
```

For space reasons, we use subscripts for the product of eight Dirac matrices.

```
(f4[μ_, ν_, ρ_, σ_, τ_, ξ_, α_, β_] =
     DiracTrace[γ[μ].γ[ν].γ[ρ].γ[σ].γ[τ].γ[ξ].γ[α].γ[β], η]) /.
           η[a_, b_] -> Subscript[η, a, b]
```

Now let us check the results. We implement the metric tensor and explicit realizations for the Dirac matrices.

```
η[i_, j_] = Which[i == j == 0, -1, i == j, 1, True, 0];

γ[0] = {{0, 0, -I, 0}, {0, 0, 0, -I}, {-I, 0, 0, 0}, {0, -I, 0, 0}};
γ[1] = {{0, 0, 0, -I}, {0, 0, -I, 0}, {0, I, 0, 0}, {I, 0, 0, 0}};
γ[2] = {{0, 0, 0, -1}, {0, 0, 1, 0}, {0, 1, 0, 0}, {-1, 0, 0, 0}};
γ[3] = {{0, 0, -I, 0}, {0, 0, 0, I}, {I, 0, 0, 0}, {0, -I, 0, 0}};
```

To check, we use all possible realizations for all indices, which means for the product of two Dirac matrices we check 16 cases, for the product of four Dirac matrices we check 256 cases, for the product of six Dirac matrices we check 4096 cases, and for the product of eight Dirac matrices we check 65536 cases.

```
Table[f1[μ, ν] - Tr[γ[μ].γ[ν]],
      {μ, 0, 3}, {ν, 0, 3}] // Flatten // Union

Table[f2[μ, ν, ρ, σ] - Tr[γ[μ].γ[ν].γ[ρ].γ[σ]],
      {μ, 0, 3}, {ν, 0, 3}, {ρ, 0, 3}, {σ, 0, 3}] // Flatten // Union

Table[f3[μ, ν, ρ, σ, τ, ξ] -
      Tr[γ[μ].γ[ν].γ[ρ].γ[σ].γ[τ].γ[ξ]],
      {μ, 0, 3}, {ν, 0, 3}, {ρ, 0, 3}, {σ, 0, 3},
      {τ, 0, 3}, {ξ, 0, 3}] // Flatten // Union

Table[f4[μ, ν, ρ, σ, τ, ξ, α, β] -
      Tr[γ[μ].γ[ν].γ[ρ].γ[σ].γ[τ].γ[ξ].γ[α].γ[β]],
      {μ, 0, 3}, {ν, 0, 3}, {ρ, 0, 3}, {σ, 0, 3},
      {τ, 0, 3}, {ξ, 0, 3}, {α, 0, 3}, {β, 0, 3}] // Flatten // Union
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

d) `identity[n]` given the identity with n variables z_k.

```
identity[n_] := With[{ν = If[EvenQ[n], n/2, (n - 1)/2]},
Product[Tanh[z[j] - z[k]], {j, 1, n}, {k, j + 1, n}] -
2^-ν/ν! Plus @@  (Function[l, Signature[l]*
  Product[Tanh[z[l[[2k - 1]]] - z[l[[2k]]]], {k, ν}]] /@
                                Permutations[Range[n]])]
```

For $n = 6$, we get the following expression.

```
n = 6;
identity[n] /. z[i_] :> Subscript[z, i]
```

To prove this expression we first use the addition theorem of the tanh function to generate all possible $\tanh(z_k)$.

```
aux1 = identity[n] /. Tanh[x_ + y_] :>
           (Tanh[x] + Tanh[y])/(1 + Tanh[x] Tanh[y]);
```

We can get rid of the denominators by multiplying with $\prod_{1 \leq k < l \leq n}^{n} (1 + \tanh(z_k) \tanh(z_l))$.

```
fac = Times @@ Flatten[Table[1 - Tanh[z[i]] Tanh[z[j]],
                             {i, n}, {j, i - 1}]];
```

Expanding now the resulting polynomial gives 0 and proves the identity under consideration.

```
Expand[(fac aux1[[1]]) + (Expand[fac #]& /@ Expand[aux1[[2]]])]
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

e) The implementation of `MultiDimensionalDet` is straightforward. Because we do not know d and n in advance, we must generate the iterators automatically. Here this is done.

```
MultiDimensionalDet[t_?(TensorRank[#] ==  Length[Dimensions[#]]&)] :=
Module[{i, n = Length[Dimensions[t]], d = Length[t], ε, part, k, l},
Sum[Evaluate[
    (* product of Levi-Civita tensors *)
    Product[ε @ Table[i[k, l], {k, d}], {l, n}]*
    (* product of matrix elements *)
    Product[part[t, ##]& @@ Table[i[k, l],
                          {l, n}], {k, d}] /. i[k_, 1] :> k],
    (* summation iterators *)
    Evaluate[Sequence @@ ({#, d}& /@
     DeleteCases[Flatten[Table[i[k, l], {l, n}, {k, d}], 1],
               i[k_, 1]])]] /. (* make Levi-Civita *)
       {ε[l:{_Integer..}] :> Signature[l], part -> Part}]
```

For 2D matrices, the results of `MultiDimensionalDet` agree with the ones from `Det`.

```
(* 2×2 matrix *)
MultiDimensionalDet[Table[Subscript[𝒯, i, j], {i, 2}, {j, 2}]]
```

```
(* 3×3 matrix *)
MultiDimensionalDet[Table[Subscript[𝒯, i, j], {i, 3}, {j, 3}]] ==
Det[Table[Subscript[𝒯, i, j], {i, 3}, {j, 3}]]
```

Here is the determinant of a $3 \times 3 \times 3$ matrix.

```
MultiDimensionalDet[Table[Subscript[𝒯, i, j, k], {i, 3}, {j, 3}, {k, 3}]]
```

For the special class of d-dimensional matrices whose elements depend only on the greatest common divisor of their indices, the multidimensional determinant is independent of the dimension [185✱]. The next input demonstrates this by using the simplest possible example, namely $a_{k_1,k_2,\ldots,k_m} = \gcd(k_1, k_2, \ldots, k_m)$.

```
Function[{o, dMax}, Table[MultiDimensionalDet @
Table[GCD @@ Table[i[j], {j, d}],
      Evaluate[Sequence @@ Table[{i[j], o}, {j, d}]]],
      {d, 2, dMax}]] @@@ {{2, 7}, {3, 4}}
```

For applications of the multidimensional determinant, see [214✱], [157✱], [160✱], [215✱], [333✱]. For noncommutative determinants, see [101✱].

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

10. Digits in π, Mediant Insertion

a) First, we generate the digits of π as a list that can be manipulated.

```
pi = RealDigits[N[Pi, 100]][[1]]
```

The explicit positions of the various digits can be obtained in this way.

```
Do[posis[i] = Flatten[Position[pi, i]], {i, 0, 9}]
```

```
??posis
```

To search for the relevant pairs, we can use pattern matching.

```
pairs[i_, i_] := Partition[posis[i], 2]

pairs[i_, j_] :=
Partition[ (* make even length *)
           If[EvenQ[Length[#]], #, Drop[#, -1]]&[ (* the positions *)
             First /@ (Sort[Join[{#, i}& /@ posis[i], {#, j}& /@ posis[j]],
                            #1[[1]] < #2[[1]]&] //.
(* look for the interesting pattern *)
  {{{_, j}, y__} -> {y},
    {x___, {y1_, k_}, {y2_, k_}, z___} -> {x, {y1, k}, z}})], 2]
```

Here are a few examples.

```
pairs[0, 0]
```

```
pairs[0, 1]
```

```
pairs[1, 0]
```

```
pairs[3, 9]
```

See [24✱] for many details concerning the relevant mathematics.

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

b) To insert a median, we first partition the original list in sublist of length 2. We keep the first element of each of the sublist and replace the second one with the median. At the end, we add the last element of the original list.

```
insertMediants[l_] :=
 Flatten[{Apply[{#1, (Numerator[#1] + Numerator[#2])/
                     (Denominator[#1] + Denominator[#2])}&,
        Partition[l, 2, 1], {1}], Last[l]}]
```

Here is an example. Starting from the list `{0,1}`, we repeatedly insert medians.

```
nl = NestList[insertMediants, {0, 1}, 6]
```

The following graphic shows the behavior of the iterated median insertion.

```
Show[Function[d, With[{l = Length[d] - 1},
      ListPlot[(* add x-coordinate *)
               MapIndexed[{(#2[[1]] - 1)/l, #1}&, d],
               DisplayFunction -> Identity, PlotJoined -> True,
               PlotStyle -> {Thickness[0.001], Hue[Random[]]}]]] /@
                                 NestList[insertMediants, {0, 1}, 12],
      DisplayFunction -> $DisplayFunction, AspectRatio -> Automatic]
```

Switching from the x,y-coordinate system to an $(x + y)$,$(y - x)$-coordinate system shows the structure slightly better.

```
Show[% /. Line[l_] :> Line[{Plus @@ #, Subtract @@ #}/Sqrt[2.]& /@ l],
     (* stretch *) AspectRatio -> 1/3, Frame -> True, Axes -> False
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

11. d'Hondt Voting

Here is a possible solution. The arguments of `dHondt` are the list `votes` of the vote counts and the list `seats` of available seats. In two positions (in the arguments and at the end of the right-hand side), we use `Condition` to check that the arguments have the correct form. In each step of `FixedPoint`, we maintain a list consisting of two lists. The first contains the number that was used to divide the vote of the current party, and the second contains the number of seats already assigned to the party. As long as enough seats are still available, we assign them. If all seats have already been assigned, nothing more happens. When the number of equal numbers is larger than is the number of remaining seats, we assign them randomly.

```
dHondt[votes_List?((Union[Head /@ #] == {Integer}) && Min[#] > 0&),
       seats_Integer] :=
FixedPoint[(* until all votes have been used *)
  Function[x,  (* the assignment *)
           Which[(Plus @@ #[[2]]) + Length[x] <= seats,
                   {MapAt[(# + 1)&, #[[1]], x], MapAt[(# + 1)&, #[[2]], x]
                 Plus @@ #[[2]] == seats, #,
                (Plus @@ #[[2]]) + Length[x] > seats,
             (* random decision *)
            CellPrint[Cell["∘ A seat is randomly assigned.", "PrintText"]];
                Function[y, {MapAt[(# + 1)&, #[[1]], y],
                             MapAt[(# + 1)&, #[[2]], y]}][x[[#]]& /@
              (Table[Random[Integer, {1, Length[x]}],
                   {(Plus @@ #[[2]]) + Length[x] - seats}])]]][
      (* the leading party *)
      Position[#, Max[#]]&[votes/#[[1]]]] &,
           {Table[1, {Length[votes]}], Table[0, {Length[votes]}]}][[2]] /;
                           (* more votes than seats *) Plus @@ votes > seats
```

First, we run the example problem.

```
dHondt[{8, 5, 9}, 6]
```

Here is another example.

```
dHondt[{31, 2, 1}, 12]
```

In the following examples, the last seat is assigned randomly.

```
dHondt[{6, 8, 9}, 6]

dHondt[{6, 8, 9}, 6]
```

For typically sized parliaments, the computation can be accomplished in a fraction of a second.

```
Timing[dHondt[{23456783, 12345732, 34897345, 7345673}, 600]]
```

As expected, this partition of the seats reflects the vote totals reasonably well.

```
(600 #/(Plus @@ #))&[{23456783, 12345732, 34897345, 7345673}] // N
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

12. Grouping, Unsorted Complements

a) Here is a simple approach; more efficient variants exist. For each list element, we first seek all nearby elements. If the resulting subsets are mutually disjointed, the problem is solved. If not, the numbers in these lists cannot be grouped in nonoverlapping disjoint subsets.

```
(* a message for the case grouping is not possible *)
group::ngr = "The numbers `1` cannot be grouped.";

group[l_, ε_] :=
Block[{groupedList =
      Union[Function[x, Select[l, (* look for all which are "near" *)
                                 (Abs[# - x] < ε)&]] /@ l]},
     groupedList /;
      If[Length[Union @@ groupedList]  ===
         Length[Join @@ groupedList], True, Message[group::ngr, l]; False]]
```

Here are three examples.

```
group[{0.01, 0.02, 0.03, 0.04, 0.05, 0.0500002}, 0.005]
```

When the numbers cannot be grouped, a message is generated.

```
group[{0.01, 0.02, 0.03, 0.04, 0.05}, 0.012]
```

Note the different choice of brackets in the following: Now all numbers fall into one class.

```
group[{0.01, 0.012, 0.013, 0.014, 0.015}, 0.1]
```

Here is a more elaborated function that finds groups of objects. Given a list p of dD vectors, the function `find⋱Groups` groups them in such a way, that within each group there exists at least one vector which has Euclidean distance less or equal to d to another vector of the same group. The function `findGroups` is written in a one-liner style and tries to achieve a good complexity by first separating clusters in each coordinate direction.

```
findGroups[p_?(MatrixQ[#, NumericQ]&), d_?NonNegative] :=
Flatten[Function[α, Apply[#&, Last /@ Rest[
(* separate all clusters *)
NestWhileList[{#[[1]], Flatten[#[[2]]]}&[
(* recursively find points of a cluster; find "chains" *)
NestWhile[Function[σ, {#[[1]], {#[[2]], σ[[2]]}}&[{Complement[σ[[1]], #],
  #}&[(* find and remove points in distance d *)
      Flatten[Fold[Function[{λ, μ},
       {Complement[λ[[1]], #], (* form nested lists, not flat ones *)
       {#, λ[[2]]}}&[Select[λ[[1]], (#.#&[#[[1]] - μ[[1]]] < d^2)&]]],
          {σ[[1]], {}}, σ[[2, 1]]][[2]]]]]], {Rest[#[[1]]], {{#[[1, 1]]}}},
    (#[[2, 1]] =!= {})&]]&, (* index all points to keep multiples *)
    {MapIndexed[C, α], {}}, #[[1]] =!= {}&]], {2}]] /@
         (* separate cluster if possible by coordinate values;
     avoid n^2 complexity in the number of points *)
        Fold[Function[{λ, δ}, Flatten[Map[RotateRight[#, δ]&,
        Split[Sort[RotateLeft[#, δ]& /@ #], #2[[1]] - #1[[1]] < d&],
         {2}]& /@ λ, 1]], {p}, Range[Length[p[[1]]]]], 1]
```

We repeat the three inputs from above. Now each element must be a vector, so, we map `List` over the above lists.

```
findGroups[List /@ {0.01, 0.02, 0.03, 0.04, 0.05, 0.0500002}, 0.005]
```

```
(* each group now has exactly one element *)
findGroups[List /@ {0.01, 0.02, 0.03, 0.04, 0.05}, 0.012]
```

```
findGroups[List /@ {0.01, 0.012, 0.013, 0.014, 0.015}, 0.1]
```

Here is a more complicated example. We use 1000 pseudorandom points from $[-1, 1]\times[-1, 1]$.

```
points = Table[N[{Cos[k], Cos[k^2]}], {k, 1000}];
```

The function `showColoredGroups` generates a graphic of the groups by connecting nearby points of a group and coloring each group.

```
(* join nearby points of each group by a line *)
makeGroupOutLine[l_, δ_] :=
Table[If[#.#&[l[[i]] - l[[j]]] < δ^2, Line[{l[[i]], l[[j]]}], {}],
      {i, Length[l]}, {j, i + 1, Length[l]}]

showColoredGroups[points_, δ_, opts___] :=
Show[Graphics[{(* the points *)
 {PointSize[0.01], GrayLevel[0.5], Point /@ points},
 (* randomly colored groups *)
 {Hue[Random[]], makeGroupOutLine[#, δ]}& /@ findGroups[points, δ]}],
    opts, PlotRange -> All, AspectRatio -> Automatic];
```

As a function of δ, we obtain one group for $\delta = 0.2$ and 40 groups for $\delta = 0.1$.

```
Show[GraphicsArray[
showColoredGroups[points, #, DisplayFunction -> Identity,
                  PlotLabel -> "δ = " <> ToString[#]]& /@
                   {0.1, 0.15, 0.2}]]
```

The next graphic shows the number of groups as a function of δ.

```
ListPlot[Table[{δ, Length[findGroups[points, δ]]}, {δ, 0, 0.2, 0.01}],
         PlotJoined -> True]
```

We end by repeating the above calculation for 1000 pseudorandom points in 3D.

```
points = Table[N[{Cos[k], Cos[k^2], Cos[k^3]}], {k, 1000}];

Show[GraphicsArray[
showColoredGroups[points, #, DisplayFunction -> Identity,
                  PlotLabel -> "δ = " <> ToString[#]]& /@
                   {0.15, 0.2, 0.3}] /. Graphics -> Graphics3D]
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

b) As a first step, we sort the given list. After that, we partition this sorted list into sublists of length two and check to see if their difference is less than *maxDiff*. If this is not the case, we have found delimiters for the groups. Knowing them, we select all pairs that form a group and join them into one list. At the end, all groups of length 1 are identified, and the groups are sorted according to their first element. Here, this approach is implemented.

```
splitInGroups[l:{_Integer..}, maxDiff_] :=
Function[l1, Sort[Join[List /@ Complement[l1, Flatten[#]], #],
                  #1[[1]] < #2[[1]]&]&[
Function[p, Map[Union[(* make groups *)
Flatten[(* take all pairs which are in one group *)
  Take[p, #]]]&, {1, -1} + #& /@ (* relevant pairs *)
  Select[Partition[Flatten[{0, Position[
     Map[(* check difference between pairs *)
         Abs[Subtract @@ #] <= maxDiff&, p, {1}], False],
                Length[l1]}], 2, 1], -Subtract @@ # > 1&]]][
     (* partition sorted list *)
        Partition[l1, 2, 1]]]][(* sort given list *)Union[l]]
```

Here are some examples.

```
splitInGroups[{1, 2, 3, 5, 6, 7, 9, 11, 22, 23}, 1]

splitInGroups[{1, 2, 3, 5, 6, 7, 9, 11, 22, 23}, 5]

splitInGroups[{1, 2, 3, 5, 6, 7, 9, 11, 22, 23}, 11]
```

Using the built-in function `Split`, it is straightforward to implement `splitInGroups`.

```
splitInGroups[l:{_Integer..}, maxDiff_] :=
           Split[Union[l], #2 - #1 <= maxDiff&]

splitInGroups[{1, 2, 3, 5, 6, 7, 9, 11, 22, 23}, 1]

splitInGroups[{1, 2, 3, 5, 6, 7, 9, 11, 22, 23}, 5]

splitInGroups[{1, 2, 3, 5, 6, 7, 9, 11, 22, 23}, 11]
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

c) The built-in `Union` called with one argument, meaning `Union[`*listOfVectors*`]` first sorts *listOfVectors* and then eliminates doubles. Because of the vector-valued nature of the elements of *listOfVectors*, vectors that are equal (in the sense of `Equal`) do not need to be adjacent after the sorting and so would not be eliminated. `Union` with an explicitly specified `SameTest`, meaning `Union[`*listOfVectors*`, SameTest -> Equal]` carries out all $n(n-1)/2$ possible comparisons between the n elements of *listOfVectors* and so has a genuine quadratic complexity. For an arbitrary transitive identification function f in `SameTest ->` f, this is the best that can be done. No sorting criterion can be derived from the identification function f in general. For the special case under consideration, real vectors that are to be identified if their components differ by less than ε, the situation is different. Here it is possible to derive a sorting function from the identification function f. Thus, it is possible to make use of the $n\ln(n)$ complexity of `Sort` and it is possible to implement a function `VectorUnion` that is faster than the built-in function `Union` with the option setting `SameTest -> Equal`.

We start by implementing a function `componentUnion` that splits a list of real vectors into groups with identical first components.

```
(* carry out unioning with respect to the first component *)
componentUnion[lists_, f_] :=
   Split[Sort[lists], f[First[#1], First[#2]]&];
```

To eliminate identical elements from a list of real vectors, we recursively split the list of vectors into groups with identical κth components. When such a group has only one element, we have a unique vector. If after splitting with respect to all d components, we have groups of vectors with more than one element this means that such a group represents one vector. We extract its first vector as a representative vector.

```
(* vector is separated *)
unionStep[{v_}, {κ_, d_}, f_] := {RotateRight[v, κ]};
(* vector is separated *)
unionStep[l_List, {κ_, d_}, f_] :=
With[{𝕗 = If[κ + 1 <= d, unionStep[#, {κ + 1, d}, f]&[
                              RotateLeft /@ #]&, Identity]},
     𝕗 /@ componentUnion[l, f]]

(* default SameTest *)
VectorUnion[lists_] := VectorUnion[lists, SameTest -> Equal]

VectorUnion[lists_, SameTest -> f_] :=
    First /@ Level[unionStep[lists, {0, Length[lists[[1]]]}, f], {-3}]
```

Now let us look at `VectorUnion` in action. The following list `L` is easy to union.

```
L = {{1, 2, 3}, {3, 4, 4}, {1, 2, 5}, {1, 2, 3}};
{VectorUnion[L], Union[L, SameTest -> Equal], Union[L]}
```

The next list `L` requires the `SameTest` option of `Union` to be specified. (Be aware that `Union[L]` returns a list with three elements.)

```
ε = $MachineEpsilon;
L = {{1 - ε, 0.}, {1., 1.}, {1 + ε, 0.}};
{VectorUnion[L], Union[L, SameTest -> Equal], Union[L]}
```

As a more complicated example for `Union` and `VectorUnion`, let us use points scattered around the vertices of a hypercube.

```
testData[n_, dim_] :=
Table[(-1)^Random[Integer] +
      Random[Real, 2 $MachineEpsilon {-1, 1}], {n}, {dim}];
```

Now, we union a list with 1000 vectors. `VectorUnion` is clearly faster.

```
data = testData[10^3, 15];

{Union[data, SameTest -> Equal] // Length // Timing,
 VectorUnion[data] // Length // Timing}
```

Due to the quadratic complexity of `Union`, `VectorUnion` can be much faster than `Union`.

```
data = testData[10^4, 16];

{Union[data, SameTest -> Equal] // Length // Timing,
 VectorUnion[data] // Length // Timing}
```

`VectorUnion` could be further optimized by unioning the components in a preprocessing step and permutating the components in such a way that most separation is done as early as possible.

Here is a small application of `VectorUnion`. We will repeatedly calculate all intersections formed by all lines through pairs of a given set of points [148★]. The function `allIntersection` calculates all nondegenerate intersections of the lines formed through the points *ps*.

```
allIntersection[ps_] :=
Module[{λ = Length[ps], sol, tab},
 tab =  Table[(* check for messages from (nearly) parallel lines *)
             sol = Check[Solve[ps[[i]] + s (ps[[j]] - ps[[i]]) ==
                                  ps[[k]] + t (ps[[k]] - ps[[l]]), {s, t}],
                         $Failed];
            (* use only finite solutions *)
            If[sol =!= $Failed && sol =!= {} && sol =!= {{}},
               ps[[i]] + s (ps[[j]] - ps[[i]]) /. sol, {}],
        (* use each line pair only once *)
        {i, λ}, {j, i + 1,  λ}, {k, i + 1, λ}, {l, k + 1, λ}];
 (* consolidate intersection points *)
 VectorUnion @ DeleteCases[Level[tab, {-2}], {}]]
```

Starting with the six vertices of a regular hexagon, in the first step, we obtain 36 different intersections out of 73 finite ones. In the second step, we obtain 18190 different intersections out of 182181 finite points after identifying nearly identical points. (The messages in the following input result from trying to calculate the intersections of two (nearly) parallel lines.)

```
nGonPoints[n_] := Table[{Cos[φ], Sin[φ]}, {φ, 0, 2Pi (1 - 1/n), 2Pi/n}];

nl = NestList[allIntersection, (* hexagon vertices *) N @ nGonPoints[6], 2];

Length /@ nl
```

Here is a visualization of all of the intersection points.

```
Show[Graphics[(* color according to generation step number *)
 MapIndexed[{PointSize[0.003 #2[[1]]], Hue[0.8 (#2[[1]] - 1)/3],
                                       Point /@ #1}&, Reverse @ nl],
     Frame -> True, PlotRange -> 3 {{-1, 1}, {-1, 1}}, Frame -> True,
     AspectRatio -> Automatic, FrameTicks -> False]]
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

d) Here is a first possible implementation. We recursively delete the first element of the first list that equals the *n*th element of the second list.

```
UnSortedComplement[l1_List, l2_List] := Fold[DeleteCases[#1, #2, {1}, 1]&,
```

Here is a simple example showing that the function `UnSortedComplement` works as expected.

```
UnSortedComplement[{1, 1, 2, 5, 5, 5, 3, 3, 2}, {1, 5, 2, 2, 3}]
```

Here is a slightly larger example, both lists having length 1000.

```
UnSortedComplement[Table[Round[10 Abs[Cos[k]]], {k, 1000}],
                   Table[Round[10 Abs[Sin[k]]], {k, 1000}]] // Timing
```

Because this implementation of the function `UnSortedComplement` has complexity $O(n_1\, n_2)$ where n_j is the length of the list l_j, forming the unsorted complement of two lists of length 50000 takes some time.

```
(ℒ1 = UnSortedComplement[Table[Round[100 Abs[Cos[k]]], {k, 50000}],
                         Table[Round[100 Abs[Sin[k]]], {k, 50000}]]) //
                                                        Length // Timing
```

We can reduce the complexity to $O(n_1 + n_2)$ by stepping through the list l_1 and using a constant-time lookup if the element occurs (including multiplicity) in l_2. Here this is implemented.

```
UnSortedComplement[l1_List, l2_List] :=
Module[{p},
          (* analyze list l2 *)
          If[Head[p[#]] === p,  p[#] = 1, p[#] = p[#] + 1]& /@ l2;
          (* step through l1 and remove the elements that occur in l2 *)
          If[Head[p[#]] === p || p[#] === 0,  #,
             p[#] = p[#] - 1; Sequence @@ {}]& /@ l1]
```

Here is again the simple test example from above.

```
UnSortedComplement[{1, 1, 2, 5, 5, 5, 3, 3, 2}, {1, 5, 2, 2, 3}]
```

For the two lists of length 1000, this version of `UnSortedComplement` uses roughly the same time as the first (it has a better complexity, but at each step it must perform more operations).

```
UnSortedComplement[Table[Round[10 Abs[Cos[k]]], {k, 1000}],
                   Table[Round[10 Abs[Sin[k]]], {k, 1000}]] // Timing
```

The example containing two 50000 element lists is already faster by more than a factor of four.

```
(ℒ2 = UnSortedComplement[Table[Round[100 Abs[Cos[k]]], {k, 50000}],
                          Table[Round[100 Abs[Sin[k]]], {k, 50000}]]) //
                                                         Length // Timing
```

The last result agrees with the above one.

```
ℒ1 === ℒ2
```

Now, we can form the unsorted complement of a still larger list in reasonable time. The next input forms the unsorted complement of two list of length 10^5.

```
(ℒ3 = UnSortedComplement[Table[Round[10^4 Abs[Cos[k]]], {k, 10^5}],
                          Table[Round[10^4 Abs[Sin[k]]], {k, 10^5}]]) //
                                                        Length // Timing
```

The resulting list exhibits some interesting structure.

```
ListPlot[ℒ3]
```

13. All Arithmetic Expressions

We use a string-oriented approach here. Suppose the numbers and the operations are given in the form of a list of strings. The following implementation does what we want.

```
allArithmeticExpressions[numbersList_List, operationsList_List] :=
(* make a Mathematica expression *)
(HoldForm @@ ToHeldExpression[StringJoin[Flatten[#]]])& /@
Union[Nest[(Sequence @@ Table[Sequence @@ Table[
(* insert the operation at all possible positions;
   keep brackets matching *)
     Insert[Delete[#, {{i}, {i + 1}}],
   StringJoin[operationsList[[j]], "[", #[[i]], ", ", #[[i + 1]], "]"],
          i], {j, Length[operationsList]}],
                    {i, Length[#] - 1}])& /@ #&,
                    {numbersList}, Length[numbersList] - 1]]
```

The idea is to enclose two neighboring numbers in parentheses and join them using one binary operation. This process is repeated for all neighboring pairs of numbers and for all given operations until only a single expression remains. For better readability of the results, we form *Mathematica* expressions via `ToHeldExpression[StringJoin[Flat`⋱
`ten[#]]])& /@ ….`

Some expressions appear twice and are eliminated using `Union`. Here is an example of the operation of `allArithmeticExpressions`. (Suppose for the moment that "𝕒", "𝕓", "𝕔", and "𝕕" are functions not yet explicitly specified.) Here are all possible expressions for three arguments and four operations.

```
allArithmeticExpressions[{"a", "b", "c"}, {"𝕒", "𝕓", "𝕔", "𝕕"}]
```

Next, we calculate all possible expressions for four arguments and two operations.

```
allArithmeticExpressions[{"a", "b", "c", "d"}, {"𝕒", "𝕓"}]
```

Here are all possible results for the four operations +, ×, gcd, and lcm and the digits of the year 1999.

```
ReleaseHold /@
 allArithmeticExpressions[{"1", "9", "9", "9"},
                              {"Plus", "Times", "GCD", "LCM"}] // Union
```

Let us give an alternative programming possibility. This time, we will manipulate expressions, not strings. We will use `ReplaceList` with a suitable rule to obtain all possible groupings.

```
allArithmeticExpressions1[args_, ops_] :=
Nest[Function[o, Flatten[Function[c, ReplaceList[c,
             {α___, β_, γ_, δ___} :> {α, #[β, γ], δ}]& /@ ops] /@ o, 2]],
                 {args}, Length[args] - 1] // Flatten
```

Using the example from above, we obtain again 32 possible expressions.

```
allArithmeticExpressions1[{a, b, c}, {𝕒, 𝕓, 𝕔, 𝕕}]
```

This approach can be easily generalized from binary to trinary operations.

```
allArithmeticExpressions2[args_, ops_] :=
Nest[Function[o, Flatten[Function[c, ReplaceList[c,
             {α___, β_, γ_, δ_, ε___} :>
             {α, #[β, γ, δ], ε}]& /@ ops] /@ o, 2]],
                 {args}, (Length[args]  - 1)/2] // Flatten
```

The next example yields 48 possible expressions.

```
allArithmeticExpressions2[{a, b, c, d, e}, {𝕒, 𝕓, 𝕔, 𝕕}]
```

One possible application for `allArithmeticExpressions` would be the automatic generation of the arithmetic games, which are popular at the beginning of each year. For an application to 4s, see [75✱], [37✱], [134✱], [72✱], and [11✱]. For all sensible compositions of vector analysis operators, see [198✱], [199✱], and [200✱].

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

14. Symbols with Values, `SetDelayed` Assignments, Counting Integers

a) The program carries out a function definition of the form `f[`*builtInName*`_] :=` *builtInName*`^2`, and then computes `f[3]`. We now let the program run.

```
names = DeleteCases[DeleteCases[Names["*"], "names"],
                        (* otherwise we might get into trouble *)
                        "RuleTable" | "$Epilog"];
```

```
(* shut off various messages *)
Off[$$Media::obsym]; Off[General::ovfl];
Off[General::under]; Off[General::unfl];
li = {};
Do[(* clear f and then give a new definition *)
   Clear[f];
   f[ToExpression[names[[i]] <> "_"]] = ToExpression[names[[i]]]^2;
   If[f[3] =!= 9,
   (* names[[i]] was not correctly treated *)
      AppendTo[li, {names[[i]], ToExpression[names[[i]]]}]],
   {i, 1, Length[names]}];
li // Length
```

Here are some of the elements of `li` shown (we select the “small” ones). (Be aware of the entry `{`*i*`, …}` in the list li. It represents the value of the iteration variable *i* from the above `Do` loop.)

```
Select[li, ByteCount[#] < 60&]
```

The list `li` contains those system functions that have a value. Note that `I` (head `Complex`) is in the list `li`, but `E` (head `Symbol`) is not. If we had carried out this operation with `SetDelayed` instead of `Set`, we would have obtained the following result.

```
li = {};
Do[Clear[f];
   ToExpression[
     "f[" <> names[[i]] <> "_] := " <> names[[i]] <> "^2"];
   If[f[3] =!= 9,
      AppendTo[li, {names[[i]], ToExpression[names[[i]]]}]],
  {i, 1, Length[names]}];
li
```

The result is shorter, but still not `{}`. We already know from an earlier exercise that problems with `Symbol` exist. The appearance of `Power` in `li` is from the special right-hand side in our function definition (the fullform is `Power[command, 2]`).

```
Clear[f];
f[power_] := power[power, 2]

f[3]

Clear[f];
f[Power_] := Power[Power, 2]

f[3]
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

b) The code returns all built-in functions *builtInFunction* that, after carrying out the definition `f[x_] :=` *builtIn*`\` *Function*`[x]`, do not result in a definition of the form `{HoldPattern[f[x_]] :>` *builtInFunction*`[x]}`. To find the functions *builtInFunction* that behave “unusually”, we build the string `"f[x_] :=` *builtInFunction*`[x]"` and convert this string into an expression. This evaluates and makes a definition for the function *f*. Then we analyze the “stringized” downvalue associated with *f*. If *builtInFunction* does not appear in the downvalue, this function will be returned. (The functions `DownValues`, `RuleDelayed`, and `List` that appear in all downvalues would have to be checked separately—but these functions work fine.)

```
Cases[{#, (* make function definition *)
 ToExpression[StringJoin["f[x_] := " <> # <> "[x]"]];
      (* analyze function definition *)
 StringPosition[ToString[FullForm[DownValues[f]]], #]}& /@
                       (* all built-in functions *) Names["System`*"], {_, {}}]
```

Three functions were returned: `Evaluate`, `Unevaluated`, and the undocumented function `Release`. In Chapters 3 and 4, we discussed the semantics of `Evaluate` and `Unevaluated`. Here they appear as the head of the second argument of `SetDelayed` and they cause the second argument to be either explicitly evaluated or avoiding any evaluation. (But because of the `HoldAll` and `SequenceHold` attribute of `SetDelayed`, this would not happen anyway.)

```
f[x_] := Evaluate[x]

??f

f[x_] := Unevaluated[x]

??f
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

c) We start by generating the lists containing the data.

```
Do[data[n] = Table[IntegerPart[k Sin[k]], {k, 10^n}], {n, 4}]
```

One way to count the numbers occurring in data would be using `Count`. We start by creating lists containing the numbers that actually occur in the data.

```
Do[occuringNumbers[n] = Union[data[n]], {n, 4}];
```

Now, we simply count how often the numbers appear.

```
With[{n = 2},
Table[{occuringNumbers[n][[i]],
       Count[data[n], occuringNumbers[n][[i]]]},
                   {i, Length[occuringNumbers[n]]}]]
```

The calculation of these numbers has a bad complexity—for each data set, many calls to `Count` have to be carried out.

```
Table[Timing[Table[{occuringNumbers[n][[i]],
          Count[data[n], occuringNumbers[n][[i]]]},
                   {i, Length[occuringNumbers[n]]}];],
      {n, 1, 4}]
```

Here is a much faster way. First, we sort the data sets. This process makes equal numbers adjacent. Then, we split them into sublists of equal numbers using the function `Split` and we determine the length of the sublists. To get measurable timings, we carry out all calculations ten times.

```
Table[Timing[Do[{First[#], Length[#]}& /@ Split[Sort[data[n]]],
                {10}]],
      {n, 4}]
```

Another possibility is to go through the list and increase a counter for every number each time it is found. This method also has a good complexity, but the absolute timings cannot compete with the last method.

```
Table[Timing[
       Do[c[occuringNumbers[n][[i]]] = 0,
         {i, Length[occuringNumbers[n]]}];
        (c[#] = c[#] + 1)& /@ data[n];
      Table[c[occuringNumbers[n][[i]]],
           {i, Length[occuringNumbers[n]]}];],
      {n, 4}]
```

Without first determining which numbers occur, we can slightly speed up the last method.

```
Table[Timing[Clear[c];
             (c[#] = If[Head[c[#]] === c, 1, c[#] + 1])& /@ data[n];
              {#[[1, 1, 1]], #[[2]]}& /@ DownValues[c];],
     {n, 4}]
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

15. `Sort[`*list*, *strangeFunction*`]`

We carry out the analysis for three arguments; the generalization to more arguments is straightforward. Here are all possible argument pairs that could be tested by `Sort`.

```
combinations = Flatten[Outer[List, {1, 2, 3}, {1, 2, 3}], 1]
```

`trueFalseCombinations` gives all possible assignments of truth values to these combinations.

```
trueFalseCombinations =
  Flatten[Permutations /@ Table[Join[Table[True, {j, i}],
                                    Table[False, {j, 9 - i}]],
                              {i, 0, 9}], 1];
```

Here are all possible lists of length 3 to be sorted.

```
allSortLists =
 Flatten[Permutations /@ Flatten[
  Table[Join[Table[1, {j, i}], Table[2, {j, 3 - i - k}], Table[3, {k}]],
        {k, 0, 3}, {i, 0, 3 - k}], 1], 1];

Short[allSortLists, 5]
```

Now, we check all possible combinations as arguments to `Sort`.

```
Do[Clear[tempSorter];
   (* make a definition for the sorting function tempSorter *)
   Set[Evaluate[tempSorter @@ #[[1]]], #[[2]]]& /@
    Thread[{combinations, trueFalseCombinations[[i]]}];
     Sort[#, tempSorter]& /@ allSortLists, {i, Length[allSortLists]}]
```

No messages were generated, so all went well.

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

16. Bracket-Aligned Formatting, Fortran Real*8, `Method` Option, Level Functions, Conversion to `StandardForm` Inputs

a) To align the brackets in a *Mathematica* expression, we will convert the expression to a string and then position each character of the string. Before dealing with the implementation of a function that aligns the square brackets, we will write a little function `restoreSpecialCharacters` that deals with special characters. `FullForm` has the annoying feature that it does not treat Greek, script, Gothic, etc. characters as characters, but rather displays them as the sequence of ASCII characters of their long names. Here this is demonstrated.

```
ToString[FullForm[Sin[α + ArcTan[β, Cot[c]]]]]

Characters[%] // InputForm
```

Instead of the last result, we would like to get

```
{"S", "i", "n", "[", "P", "l", "u", "s", "α",",", " ",
 "A", "r", "c", "T", "a", "n", "[", "β", "]", ",", " ",
 "C", "o", "t", "[", "c", "]", "]", "]", "]"}
```

The function `specialCharacter` converts a list of characters representing a special character into the corresponding special character. We define `specialCharacter` for all available special characters.

```
Apply[Set[specialCharacter[#1], #2]&,
{Characters[StringDrop[StringDrop[
  ToString[FullForm[#]], -2], 3]], #}& /@
  DeleteCases[Select[(* all characters *)
  FromCharacterCode /@ Range[10^5],
         Characters[ToString[FullForm[#]]][[-2]] === "]"&], "]"], {1}];
```

The next input shows `specialCharacter` at work for the character α.

```
specialCharacter[{"A", "l", "p", "h", "a"}] // InputForm
```

Using `specialCharacter`, it is straightforward to write a function `restoreSpecialCharacters`, which restores all special characters in a list of characters. We recognize the beginning of a special character by the appearance of `"\\"`.

```
restoreSpecialCharacters[stringList_] :=
Module[{slashPosis, specialCharacterPosis, newCharacters},
 (* position of a \ indicating a special character *)
 slashPosis = Position[stringList, "\\"];
 (* position of the special character characters *)
 specialCharacterPosis =
 Table[k = #[[1]] + 1;
       While[stringList[[k]] =!= "]",
             k = k + 1]; {#[[1]], k}]& /@ slashPosis;
 (* the to be substituted character *)
 newCharacters = specialCharacter[Take[stringList,
             # + {2, -1}]]& /@ specialCharacterPosis;
 (* the position of repeated replacements to be done *)
 posisData = MapIndexed[(First[#1] - Last[#1])&,
                 Transpose[{specialCharacterPosis,
                 Drop[FoldList[Plus, 0,
                     -Apply[Subtract, specialCharacterPosis, {1}]], -1]}]];
 (* do the exchange of characters *)
 Fold[Insert[Delete[#1, List /@ (Range @@ #2[[1]])],
          #2[[2]], #2[[1, 1]]]&,
          stringList, Transpose[{posisData, newCharacters}]]]
```

Here is the function `restoreSpecialCharacters` applied to the above expression that contained the special characters α and β.

```
StringJoin @ restoreSpecialCharacters[
                   Characters[ToString[FullForm[Sin[α + Cos[β]] + γ]]]]
```

Now, we can implement the function `AlignBrackets`. Its argument is a *Mathematica* expression. `alignBrack⋱ets` writes a cell that contains the `FullForm` of this expression in a properly aligned way. The two auxiliary functions `indexList` and `prefaceSpaces` index the elements of a list and prepend white space to a list.

```
indexList[{l___, c:C[i_, j_, _]}] :=
With[{λ = Length[{l}]},
     Append[MapIndexed[C[i, -λ + #2[[1]] + j - 1, #1]&, {l}], c]]

prefaceSpaces[{c:C[i_, j_, _], r___}] :=
         Join[Table[C[i, k, " "], {k, j - 1}], {c}, {r}]
```

The implementation idea behind `alignBrackets` is simple: The function `AlignBrackets` starts by generating a string of the `FullForm` of *code*. The opening and closing square brackets in this string are then located and positioned. Keeping the relative position of these characters, we position of the square brackets. Then, we position all other characters accordingly.

```
(* AlignBrackets is a formatting function ---
   avoid any evaluation *)
SetAttributes[AlignBrackets, HoldAllComplete];

AlignBrackets[code_] :=
Module[{characters1, characters, row, column, markedBrackets,
        CPosis, lines, indexedLines, minColumn, indentedIndexedLines,
        indentedFullyIndexedLines, finalLines, cellString},
(* transform unevaluated input into characters *)
characters1 = Characters[ToString[FullForm[HoldComplete[code]]]];
characters = Drop[Drop[characters1, 13], -1];
(* restore special characters *)
characters = restoreSpecialCharacters[characters];
(* mark positions of opening and closing square brackets;
   one at each line and new "[" indented *)
row = 0; column = 0;
markedBrackets =
 Which[# === "[", C[row = row + 1, column = column + 1, #],
       # === "]", C[row = row + 1, column = column - 1;
                  column + 1, #],
       True, #]& /@ characters;
(* put a "," after a "]" on the same line *)
markedBrackets =
markedBrackets //. {a___, C[i_, j_, "]"], ",", b___} :>
                   {a, "]", C[i, j + 1, ","], b};
(* position of marked characters *)
CPosis = Flatten[{0, Position[markedBrackets, _C]}];
(* split into lines *)
lines = Take[markedBrackets, {#[[1]] + 1, #[[2]]}]& /@ Partition[CPosis, 2,
(* position all characters of one line *)
indexedLines = indexList /@ lines;
(* left-most column *)
minColumn = Min[#[[2]]& /@ Cases[indexedLines, _C, {2}]];
(* left-most column is left flush *)
indentedIndexedLines = indexedLines /. C[i_, j_, s_] :>
                              C[i, j - minColumn + 1, s];
(* add " " to the left *)
indentedFullyIndexedLines =  prefaceSpaces /@ indentedIndexedLines;
(* add new line at the end of each line *)
finalLines = Append[Last /@ #, FromCharacterCode[10]]& /@
                                indentedFullyIndexedLines;
(* form one string *)
cellString = StringDrop[StringJoin[Flatten[finalLines]], -1];
(* display the string *)
CellPrint[Cell[cellString, "Input", FontWeight -> "Plain"]]]
```

Now, let us test the function `AlignBrackets`. Here is a simple nested input. It is easy to check the alignment by inspection. Note that the first character of the first line has to be indented to achieve the overall alignment structure needed.

```
AlignBrackets[Sin[α + n + 𝔽[b, D[c[g], g], 1 + 1]]]
```

Here is a test that the last expression is correct—we evaluate the last cell generated.

```
SelectionMove[SelectedNotebook[], Previous, Cell, 3];
SelectionEvaluateCreateCell[SelectedNotebook[]]
```

The next expression looks very symmetric after the alignment of the square brackets.

```
AlignBrackets @@ {Nest[f, x, 8]}
```

The last example shown here is the formatted version of the function `RotatedBlackWhiteStrips` from Subsection 1.1.2.

```
AlignBrackets[
Graphics[MapIndexed[{If[(-1)^(Plus @@ #2) == 1,
               GrayLevel[0], GrayLevel[0.8]],
       Polygon[Join[#1[[1]], Reverse[#1[[2]]]]]}&,
            Partition[Partition[
              Distribute[{N[{{+Cos[#], Sin[#]},
                             {-Sin[#], Cos[#]}}]& /@
                               Range[0, 2 Pi, 2Pi/q],
                          N[(1 - (#/(2 Pi)))*
                             {Cos[ρ #], Sin[ρ #]}]& /@
                                 Range[0, 2 Pi, 2 Pi/p]},
     List, List, List, Dot], p + 1], {2, 2}, 1], {2}],
          AspectRatio -> Automatic, PlotRange -> All]]
```

We leave it to the readers to refine the function `alignBrackets` for the case of long lists of arguments, for the case that the expression contains strings, and to adapt details it to their formatting preferences.

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

b) Here is one suggestion.

```
FortranReal8[n_Integer] :=
If[n === 0, "0.D0",
   If[Sign[n] === -1, "-", ""] <> "0." <> StringJoin[ToString /@
   FixedPoint[If[Last[#] === 0, Drop[#, -1], #]&, #]] <>
   "D" <> ToString[Length[#]]&[IntegerDigits[n]]]
```

We now give three examples.

```
FortranReal8[18936]
```

```
FortranReal8[-3]
```

```
FortranReal8[0]
```

Much broader Fortran transformation utilities can be found in the C, FORTRAN77 and other formats code generation package by M. Sofroniou (*MathSource* 0205-254) and Fortran definitions by P. Janhunen (*MathSource* 0202-172).

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

c) Finding the 15 built-in functions that have a `Method` option is straightforward.

```
functions = ToExpression[#, InputForm,
              Unevaluated]& /@ DeleteCases[Names["*"], "I"];
```

```
functionsWithOptions = ToString /@ Select[functions,
                                    MemberQ[Options[#], Method, {-1}]&]
```

Finding the possible options settings within *Mathematica* is more tricky. Unfortunately, there is not a `PossibleOptionSettings[`*function*`,` *option*`]` command, so that we cannot successfully evaluate `PossibleOptionSettings[NDSolve, Method]`, etc. The best one can do within *Mathematica* is to have a close look at the messages of the functions. Maybe a usage message of a function will say what are the possible settings, or maybe a warning message issued when an unknown option setting is used, contains some hints about the allowed settings. So, we load all messages.

```
(* load all messages (usage and warning/error messages) *)
Get[ToFileName[{$TopDirectory, "SystemFiles", "Kernel",
                "TextResources", $Language}, #]]& /@
    {"Messages.m", "Usage.m"};
```

Extracting the messages that contain the word *method* yields 55 messages that might contain some hints on possible settings.

```
Off[Message::name];
potentiallyUsefulMessages = DeleteCases[
Flatten[Select[Messages[#], (* match method or Method *)
       (Or @@ ((StringMatchQ[#, "*Method*"] ||
                StringMatchQ[#, "*method*"])& /@
                Cases[#, _String, {-1}]))&]& /@ functions],
 RuleDelayed[Verbatim[HoldPattern][MessageName[Method, _]], _]];

potentiallyUsefulMessages // Length
```

We now investigate the content of the messages. For a programmatic treatment, we would like to avoid reading the messages. So, without implementing a limited version of artificial intelligence, the best is to just search for built-in names and numbers in the texts.

```
functionsFromText[s_String] :=
DeleteCases[
Select[StringTake[s, #]& /@ Partition[(* make words *)
          Flatten[{1, {-1, +1} + #& /@ First /@
             StringPosition[s, {" ", ".", ",", ";"}],
        StringLength[s]}], 2], (* extract built-in functions *)
       (Context[#] === "System`" ||
        Head[ToExpression[#]] === Integer) &], "Method"]
```

We arrive at the following set of built-in functions that are referred to by the 15 functions that have a `Method` option.

```
Off[Context::notfound]; Off[ToExpression::sntx]; Off[ToExpression::sntxi];
data = Union[Flatten[{ToString[#[[1, 1, 1]]],
       functionsFromText[#[[2]]]}]]& /@ potentiallyUsefulMessages;
```

Consolidating the result and eliminating all functions that are themselves options, as well as some obvious nonoption settings, yields the following conjectured `Method` option settings.

```
Off[First::normal];
allOptions = ToString /@ Union[First /@ Flatten[Options /@ functions]];

functionsWithOptions = ToString /@ Select[functions,
                            MemberQ[Options[#], Method, {-1}]&]

functionsWithAnyOption = ToString /@ Select[functions, Options[#] =!= {}&];
```

```
Off[Attributes::notfound];
functionsAndPotentialMethodSettings =
DeleteCases[{#[[1]], DeleteCases[
 If[Union[LetterQ /@ #[[2]]] === {False, True},
               (* no mixed number symbol settings *)
    DeleteCases[#[[2]], _?(Not[LetterQ[#]]&)], #[[2]]],
    (* options and option settings have mostly different names *)
    _?(MemberQ[Join[allOptions, functionsWithOptions,
                   functionsWithAnyOption], #]&)]}& /@
DeleteCases[Function[f, {f, DeleteCases[
  Union[Flatten[Select[data, MemberQ[#, f]&]]], f]}] /@
                                  functionsWithOptions,
 (* these surely are not Method option settings *)
  "Value" | "For" | "If" | "Not" | "With" | "Infinity" |
  _?(MemberQ[Attributes[#], NumericFunction]&), {-1}], {_, {}}];

Print /@ functionsAndPotentialMethodSettings;
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

d) If a function takes a level specification, then its usage message will say so. We start by loading and collecting all usage messages.

```
Get[ToFileName[{$TopDirectory, "SystemFiles", "Kernel",
                "TextResources", $Language}, "Usage.m"]];

systemCommands = Names["System`*"];
(* clear the ReadProtected attribute *)
If[MemberQ[Attributes[#], ReadProtected],
   ClearAttributes[#, ReadProtected]]& /@
     Apply[Unevaluated, ToHeldExpression /@
              DeleteCases[systemCommands, "I"], {1}];
(* make list of all messages *)
allMessages = (Messages @@ #)& /@ (ToHeldExpression[#]& /@
                                 DeleteCases[systemCommands, "I"]);

Off[Part::partw];
allUsageMessages = Select[allMessages, #[[1, 1, 1, 2]] === "usage"&];
```

Next, we extract all messages that contain explicitly the word "level".

```
messagesContaing["level"] =
Select[#[[1, 2]]& /@ allUsageMessages,
       StringMatchQ[#, "*level*"] && StringMatchQ[#, "*[*"]&];

messagesContaing["level"] // Length
```

Here is one example.

```
messagesContaing["level"][[11]]
```

Without explicitly reading all messages and deciding if these functions take a level specification, we will extract from the body of the messages all that contain explicit argument specifications of the form *function*[*args*, *levelspecification*, *other args*].

```
goLeft[s_String, pos_] :=
Module[{p = pos},
(* go to the left until function name starts *)
 While[p > 0 && Not[StringTake[s, {p, p}] === " "],
       p = p - 1]; p + 1]
```

```
getMathematicaExpression[s_String] :=
Select[StringTake[s, {goLeft[s, #[[1]]], #[[2]]}]& /@
  (* position of function arguments *)
 Partition[Union[Flatten[{StringPosition[s, "["],
                          StringPosition[s, "]"]}]], 2],
 (* "lev" appears somewhere *)
 StringMatchQ[#, "*lev*"]&]
```

Here are the functions that were found.

```
Flatten[getMathematicaExpression /@
            messagesContaing["level"]] // TableForm
```

But unfortunately, not all usage message bodies contain "lev" explicitly. To find the remaining ones, like `Outer`, we would have to refine the textual analysis of the message body.

```
??Outer
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

e) The easiest way to achieve the conversion would be to use the menu item Cell⟶Display As⟶StandardForm. While this would generate `StandardForm` formatting, the resulting spacing would be suboptimal. Using the menu item Cell⟶Convert To⟶StandardForm would give less white space and proper `StandardForm` characters, but we would loose all comments. So, we implement a string-manipulation based approach to modify the `InputForm` cells before we will display them as `StandardForm` cells.

The first argument of the input form cells of the *GuideBooks* are all either strings of compound expressions with the head `TextData`. As a first step, we form pure strings of all input form cell bodies by removing the style boxes for comments.

```
makeOneString[s_String] := s

makeOneString[TextData[sb_StyleBox]] := makeOneString[TextData[{sb}]]

makeOneString[TextData[l_List]] :=
  StringJoin[Which[Head[#] === String, #,
                   Head[#] === StyleBox, #[[1]],
                   True, Print["Unexpected item: ", #]; ""]& /@ l]
```

To preserve the alignment, we replace multiple white spaces with twice as many white spaces (`StandardForm` uses a smaller natural white space size). Single white spaces, we do not change.

```
addMultiSpaces[s_String] :=
Module[{chars, groupedChars, newChars1, newChars2},
    chars = Characters[s] /. {"\t" -> Sequence @@ Table[" ", {3}]};
    (* group multiple white space together *)
    groupedChars = Split[chars, (#1 == " " && #2 == " ")&];
    newChars1 = Flatten[If[Length[#] === 1, #, Join[#, #]]& /@ groupedChars
    newChars2 = Flatten[If[# === "\n", {"\n", " "}, #]& /@ newChars1];
    StringJoin[newChars2]]
```

Next, we deal with using special symbols. The function `useShortName` uses the short name *newName* instead of the old name *oldName* when the left and right neighboring characters come from the lists *goodLeftCharacters* and *good RightCharacters*.

```
useShortName[s_String, {oldName_, newName_},
              {goodLeftCharacters_, goodRightCharacters_}] :=
Module[{s1 = " " <> s <> " ", chars, thePositions, goodPositions},
       chars = Characters[s1];
       thePositions = StringPosition[s1, oldName];
       (* find isolated occurrences *)
       goodPositions = Select[thePositions,
              (MemberQ[goodLeftCharacters , chars[[#[[1]] - 1]]] &&
               MemberQ[goodRightCharacters, chars[[#[[2]] + 1]]])&];
       StringTake[StringReplacePart[s1, newName, goodPositions], {2, -2}]]
```

For the quantities `Pi`, `I`, `E`, and `Infinity`, we allow the surrounding characters to be white space and numbers to the left.

```
useShortIPiInfinity[s_String] :=
  Fold[useShortName[#1, #2, {{"{", "(", ",", " ", "\n", ";", "/",
                              "1", "2", "3", "4", "5","6","7","9","0"},
                             {"}", ")", ",", " ", "\n", ";", "^"}}]&,
       s, {{"Pi", "π"}, {"I", "ⅈ"}, {"E", "ⅇ"}, {"Infinity", "∞"}}]
```

Here is an example of the transformations that `useShortIPiInfinity` carries out.

```
useShortIPiInfinity["NameWithPiInside + Pi + 1 + DirectedInfinity[2] -
                     Sum[k, {k, Infinity}] + 4 u I + 2 I I"]
```

For the operators ==, `->`, and `:>`, we assume white space characters as neighbors.

```
useShortEqualRule[s_String] :=
  Fold[useShortName[#1, #2, {{" ", "\n"}, {" ", "\n"}}]&,
       s, {{"==", "⩵"}, {"->", "→"}, {":>", "⧴"}}]
```

For a more typical `StandardForm` appearance, we also replace double brackets from `Part`, namely `[[` and `]]` by 〚 and 〛.

```
usePartBrackets[s_String] :=
Module[{s1, theOpeningPositionsAll, λ,
        literalStringPositions, theClosingPositions,
        theOpeningPositionsInsideStrings, theOpeningPositions,
        𝕔, sℓ, (* count intermediate brackets *) 𝕠𝕔},
       chars = Characters[s1 = " " <> s <> " "];
       (* the opening Part double brackets;
    (assume they always occur together and not on separate lines *)
       theOpeningPositionsAll = StringPosition[s1, "[["];
       (* inside strings, do not replace double brackets *)
       literalStringPositions = Partition[First /@ StringPosition[s1, "\""]
       theOpeningPositionsInsideStrings =
       Select[theOpeningPositionsAll, (Or @@
          (Function[{l, u}, l < #[[1]] < u] @@@ literalStringPositions))&];
       theOpeningPositions = DeleteCases[theOpeningPositionsAll,
             Alternatives @@ theOpeningPositionsInsideStrings];
       λ = Length[theOpeningPositions];
       sℓ = StringLength[s1];
       (* find closing Part double brackets *)
       theClosingPositions =
       Table[𝕔 = theOpeningPositions[[k]] + 2; 𝕠𝕔 = 0;
             (* step through the string until closing pair is found *)
             While[(StringTake[s1, 𝕔] =!= "]]" || 𝕠𝕔 =!= 0) && Max[𝕔] <= sℓ,
                   (* find matching pair in case of nesting *)
                   If[StringTake[s1, 𝕔[[1]] {1, 1}] == "[", 𝕠𝕔 = 𝕠𝕔 + 1];
                   If[StringTake[s1, 𝕔[[1]] {1, 1}] == "]", 𝕠𝕔 = 𝕠𝕔 - 1];
                    𝕔 = 𝕔 + 1]; If[Max[𝕔] > sℓ, $Failed, 𝕔], {k, λ}];
       If[MemberQ[theClosingPositions, $Failed],
          (* return original string in case of unmatched brackets *) s,
          StringTake[StringReplacePart[s1, Join[Table["〚", {λ}], Table["〛",
                     Join[theOpeningPositions, theClosingPositions]], {2, -
```

To adjust for the smaller width of the characters ⩵, →, and :→ instead of ==, ->, and :>, we add some white space in the lines following occurrences of these characters.

```
adjustIndentation[s_String] :=
Module[{newLinePositions, lineStrings, lineCharacters,
        counts, cCounts, newNewLines},
       newLinePositions = StringPosition[s, "\n"];
       lineStrings = StringTake[s, # + {1, 0}]& /@
           Partition[Flatten[{0, newLinePositions, StringLength[s]}], 2];
       lineCharacters = Characters /@ lineStrings;
       (* count narrower characters *)
       counts = Count[#, "π" | "→" | ":→" | "⩵"]& /@ lineCharacters;
       cCounts = Rest[FoldList[Plus, 0, counts]];
       newNewLines = StringJoin["\n", StringJoin[Table[" ", {#}]]]& /@
                                                     Drop[cCounts, -1];
      (* add white space to the newline characters *)
      StringReplacePart[s, newNewLines, newLinePositions]]
```

The last function could be extended to analyze more carefully the number of new characters above white space characters of following lines.

Now, we have all individual transformations together to define the function `makeStandardFormCell` that converts a formatted `InputForm` cells to a `StandardForm` cell with similar indentation. We use the style `"RigidStandardFormInput"` which is defined in the stylesheet of the *GuideBooks*. We avoid formatting powers, sums, products, integrals and so on in a truly 2D manner.

```
makeStandardFormCell[c:Cell[expr_, "Input", rest___]] :=
Module[{s1, s2, s3, s4, s5, s6},
       If[MemberQ[expr, _BoxData, Infinity], c,
          (* carry out all of the above transformations *)
          s1 = makeOneString[expr];
          s2 = addMultiSpaces[s1];
          s3 = useShortIPiInfinity[s2];
          s4 = useShortEqualRule[s3];
          s5 = usePartBrackets[s4];
          s6 = adjustIndentation[s5];
          Cell[BoxData[s6], "RigidStandardFormInput", rest]]]

makeStandardFormCell[c_] := c
```

Here is a simple example. This is the formatted original `InputForm` cell.

```
E^(2 + 2 I pi) == E^2 + 1/Infinity + {{0}}[[1, {0}[[1]] + 1]] /.
                  pi -> Pi /. E^2 :> ⅇ2
```

Here we create a formatted version of the corresponding `StandardForm` cell.

```
CellPrint @ (sfCell = makeStandardFormCell @
(* the underlying cell expression of the last input *)
Cell["\<\
E^(2 + 2 I pi) == E^2 + 1/Infinity + {{0}}[[1, {0}[[1]] + 1]] /.
                  pi -> Pi /. E^2 :> ⅇ2\
\>", "Input"])
```

We could also remove all formatting and let the *Mathematica* front end do all formatting, including adding spaces and linebreaks. While this will remove all alignments, for smaller inputs such a formatting is sometimes preferable. The function `removeWhiteSpace` removes all white space that has no semantic meaning. For these cells, we use the cell style "`StandardFormInput`".

```
removeWhiteSpace[body_] :=
 FixedPoint[StringReplace[#,
          (* remove white space around low-binding operators *)
          {" // " -> "//", " + " -> "+", " - " -> "-",
           " = " -> "=",  " := " -> ":=", " → " -> "→", " ⧴ " -> "⧴",
           " ⩵ " -> "⩵", " != " -> "!=", " === " -> "===", " =!= " -> "=!=",
           " > " -> ">", " < " -> "<", " >= " -> ">=", " <= " -> "<=",
           " /. " -> "/.", " //. " -> "//.",  " /; " -> "/;",
           " /@ " -> "/@", " //@ " -> "//@", " @@ " -> "@@",
           " @@@ " -> "@@@", " && " -> "&&", " || " -> "||", " | " -> "|",
           ", " -> ",", "; " -> ","}]&,
           (* condense multiple whitespace *)
           FixedPoint[StringReplace[#, {"\n" -> " ", "  " -> " "}]&,
                      body]]

(* remove white space in strings and use other cell style *)
makeAutomaticallyFormattedStandardFormCell[expr_] :=
expr //. c:Cell[BoxData[b_], "RigidStandardFormInput", r___] :>
           Cell[BoxData[removeWhiteSpace[b]], "StandardFormInput", r]
```

Here is the example cell from above with the alternative formatting.

```
CellPrint @ makeAutomaticallyFormattedStandardFormCell[sfCell]
```

Now, we can use the function `makeStandardFormCell` to convert all cells of a whole *GuideBook* notebook.

```
makeNotebookWithStandardFormCells[Notebook[cells_, rest___]] :=
Module[{physicalCells, newCells},
 physicalCells = Flatten[cells //. Cell[CellGroupData[l_, ___], ___] :> l];
 newCells = makeStandardFormCell /@ physicalCells;
 Notebook[newCells, rest]]
```

Here is an example. We use this notebook.

```
notebooksTMGBs = Flatten[
     {Function[{c, n}, (c <> ToString[#] <> ".nb")& /@ Range[n]] @@@
      {{"1_Programming_", 6}, {"2_Graphics_", 3},
       {"3_Numerics_", 2}, {"4_Symbolics_", 3}}}];

fileNames = ToFileName[ReplacePart["FileName" /.
  NotebookInformation[EvaluationNotebook[]], #, 2]]& /@ notebooksTMGBs;

(* read in the stylesheet *)
stylesheet = Get[ToFileName[ReplacePart["FileName" /.
                  NotebookInformation[EvaluationNotebook[]],
                            "GuideBooksStylesheet.nb", 2]]];

(* read in this notebook *)
nb = Get[fileNames[[6]]];

(* reformat inputs *)
sfCellNb = makeNotebookWithStandardFormCells[nb];
```

When displaying the reformatted notebook `sfCellNb` in the front end, the strings inside the `BoxData` are automatically converted into (nested) box structures. (Most inputs of the *GuideBooks* will evaluate in such a newly formatted notebook in the same manner as they did in the original notebook. But some inputs analyze the structure of a notebook and as a result might behave differently or not work properly.)

```
(* To view the nb generated with the following input
  properly, the stylesheet GuideBooksStylesheet.nb
  should be assigned.
  For saving the notebook in another directory,
  no private stylesheet should be embedded
  and the stylesheet should be in the other directory.
  The new notebook should be closed, opened again and saved again
  preserve the boxes generated by the front end. *)
NotebookPut[sfCellNb /. (StyleDefinitions -> _) ->
                        (StyleDefinitions -> stylesheet)]
```

And here is a version with all alignments and white spaces removed.

```
sfCellNbC = makeAutomaticallyFormattedStandardFormCell[sfCellNb];

NotebookPut[sfCellNbC /. (StyleDefinitions -> _) ->
                         (StyleDefinitions -> stylesheet)]
```

```
NotebookPut[makeNotebookWithStandardFormCells[#]]& /@ fileNames
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

17. `ReplaceAll` Order, Pattern Realization, Pure Functions

a) This is the original function `orderedTriedExpressions`.

```
orderedTriedExpressions[expr_] :=
Module[{bag = {}},
       expr /. x_ :> Null /; (AppendTo[bag, x]; False);
       bag]
```

To compare the results of the functions `orderedTriedExpressions`*i* with the result of `orderedTriedExpressions`, we will use the following test expression *expr*.

```
expr = {a[A[B]][C], {b, {c, d, Sin[ArcTan[1, e]]}}};
res = orderedTriedExpressions[expr]
```

It is straightforward to implement a version of `orderedTriedExpressions` that uses only built-in functions. Instead of the variable `bag`, we just use any built-in function. (To avoid the creation of a nonbuilt-in function ...`$`*i*, we use `Block` instead of `Module` and `Factor` instead of `x` and `Expand` for `bag`.)

```
orderedTriedExpressions2[expr_] :=
Block[{Expand = {}},
       expr //. Factor_ :> Null /; (AppendTo[Expand, Factor]; False);
       Expand]

orderedTriedExpressions2[expr] === res
```

By using a pure function, we eliminate the pattern variable `expr`.

```
orderedTriedExpressions3 =
Block[{Expand = {}},
       # //. Factor_ :> Null /; (AppendTo[Expand, Factor]; False);
       Expand]&;

orderedTriedExpressions3[expr] === res
```

The implementation of a version of `orderedTriedExpressions` without assignments is slightly more complicated. `ReplaceAll` will try a subexpression and, if no match occurs, will try the head of the expression and its elements. If no match occurs in any of them, `ReplaceAll` will recursively continue. We can get a list of head and arguments from `Level[`*subExpression*`, {1}, Heads -> True]`. To achieve the recursive treatment of all subexpressions without using assignments, we use a self-reproducing pure function via `#0`. We end the recursion when we encounter an atomic expression. Putting all of this together results in the following implementation. (The `Sequence @@` ... destroys unnecessary outer lists.)

```
orderedTriedExpressions4 =
{(Sequence @@ {#, If[AtomQ[#], Sequence @@ {}, Sequence @@ (#0 /@
                       DeleteCases[Level[#, {1}, Heads -> True], {},
                                   {1}])]})&[#]}&;
```

Here is a check that `orderedTriedExpressions4` works correctly.

```
orderedTriedExpressions4[expr] === res
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

b) The idea for the function `PatternRealization` is as follows. First, we look for all pattern variables (including `Pattern`) in *expressionWithPatterns* (by using `Position[Hold[`*expressionWithPatterns*`], Pattern]`). Then we carefully extract the first arguments of all `Pattern`s with `HeldPart` and collect them in a list (after eliminating multiple elements). Then we define an auxiliary function `faux` whose arguments are the same as those of *expression* and whose right-hand side is just the list of the pattern variables. After applying this function to the arguments of *expression*, the result is a list of the actual realizations of the pattern variables. Finally, we combine corresponding pattern variables and realizations. To avoid evaluation of any variable, we use `HoldForm` everywhere in the result; this has the convenient side effect that `Sequence`s arising from the pattern matching are also displayed. We assume that the first argument of `PatternRealization` is free of patterns. Here is the corresponding code.

```
(* to avoid any evaluation of the argument *)
SetAttributes[PatternRealization, HoldAllComplete];

PatternRealization[expr_, form_] :=
Module[{faux, allPatternVars, lhs, rhs},
(* the local function *)
SetAttributes[faux, HoldAll];
(* all pattern variables *)
allPatternVars = List /@ Union[Join @@ Apply[HeldPart[Hold[form], ##]&,
   Append[Drop[#, -1], 1]& /@ Position[Hold[form], Pattern], {1}]];
(* define local function with same pattern *)
Set @@ {Apply[faux, Hold[form], {1}][[1]], allPatternVars};
(* prepare left hand side for output *)
lhs = List @@ Apply[HoldForm, allPatternVars, {1}];
(* apply the function faux and prepare right hand side for output *)
rhs = List @@ Apply[HoldForm, Apply[faux, Hold[expr], {1}][[1]], {1}];
(* merge corresponding patterns and realizations *)
Apply[RuleDelayed, Transpose[{lhs, rhs}], {1}]] /;
                         (* usable only if expr matches form *)
                         (MatchQ[Hold[expr], Hold[form]])
```

Here are three examples.

```
PatternRealization[f[1, 2, 3, 4, {1, 2}], f[x_, y__, z:{1, 2}]]

PatternRealization[g[w[4], 5], g[x_:2, y:_w, z_Integer]]

PatternRealization[g[1, 2, 3], g[HoldPattern[x__Integer]]]
```

Note that a `Sequence` of matches is displayed as `HoldForm[`*sequence*`]`.

To write a purely functional form of `PatternRealization` (called `PatternRealizationF`), we have to get rid of the variables `faux`, `allPats`, `lhs`, and `rhs`. The last three are easily eliminated by using pure functions. To get rid of `faux`, we change the implementation slightly; we do not apply a named function to the arguments of *expression*, but this time apply a replacement rule, which has the same effect as `faux` in the implementation above. So, we can implement as follows.

```
SetAttributes[PatternRealizationF, HoldAllComplete];

PatternRealizationF[expr_, form_] :=
(Apply[RuleDelayed, Transpose[{List @@
      Apply[HoldForm, #1, {1}], (Hold[expr] /. #2)[[1]]}], {1}]& @@
 ({#1, RuleDelayed @@ {HoldPattern[form],
    List @@ Apply[HoldForm, #1, {1}]}}&[
      List /@ Union[Join @@ Apply[HeldPart[Hold[form], ##]&,
         Append[Drop[#, -1], 1]& /@
            Position[Hold[form], Pattern], {1}]]])) /;
                         (MatchQ[Hold[expr], Hold[form]])
```

Again, four examples follow.

```
PatternRealizationF[h[1, 2, 1, 2, 3, 3], h[x__, x__, z__?(# > 2&)]]

PatternRealizationF[k[2, 2, 2, 2], k[a:(1 | 2), b:(2)..]]

PatternRealizationF[H[1, 1], H[a:(b:(c:(d:((e_)..))))]]

PatternRealizationF[H[1, 2, 3], HoldPattern[H[x__, HoldPattern[y_]]]]
```

The function PatternRealizationF can be considerably improved, especially for wrappers like Verbatim appearing as arguments.

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

c) Unfortunately, we cannot simply do a simple replacement like the following.

```
rule[x_] := Function[body_] :>
                 With[{newBody = body /. Slot[1] -> x}, Function[x, newBody]]
```

The following result is obviously not, what we want.

```
(#^2&[#]&) /. rule[η]
```

The problem is that the *body* /. Slot[1] -> *x* replacement does not properly take into account the scoping range of the Function under consideration. In addition, the pure function might take more than one argument. Let us deal with the number of arguments first. numberOfSlots determines how many arguments a body of a pure function that uses Slots expects. Note that not all of the Slots might actually be in use later.

```
SetAttributes[numberOfSlots, HoldAll];

numberOfSlots[body_] :=
Function[vars, (* how many Slots were used *)
                  Max[Position[Position[(* which Slots are used *)
                     Hold[body]&[Sequence @@ vars],
                        #]& /@ vars, _?(# =!= {}&), {1},
                           Heads -> False]]][Table[Unique[x],
                     (* maximal number of Slots *)
                     {Max[First /@ Cases[Hold[body], Slot[_], {-2}]]}]]
```

Here are two examples.

```
{numberOfSlots[#[3]^2&[#]], numberOfSlots[#3^2&[#1, #4]]}
```

Now, we deal with the replacement of the Slots within the correct scoping range. rule is a Rule that does the actual replacement of the one-argument pure functions using Slots with pure functions that use explicit variables. We first determine how many named variables are needed (using the function numberOfSlots from above). Then we generate a list of unique variables names and plug the new body of the pure function (with a named variable instead of a Slot) into a pure function of the form Function[*listOfNewVariables*, *newBody*]. To make sure that the Slots replaced by named variables have the correct scoping radius, we evaluate a held version of the pure function and check if no free Slots remain using the condition FreeQ[*newBody*, Slot, {-1}, Heads -> True]. We create unique dummy variables using Unique, which makes sure that the dummy function variable is not independently occurring in the body of Function.

```
rule[x_] =
  Function[body_?((* check if Slots are present *)
                Function[b, MemberQ[Hold[b], Slot, {-1}, Heads -> True],
                         {HoldAll}])] :>
       With[{(* new body with named vars *)
             newBody = Module[{vars = Table[Unique[x],
                                    {numberOfSlots[body]}], F},
             (* a dummy head *)
             SetAttributes[F, HoldAll];
             (* construct the new pure function *)
             DeleteCases[(function[vars, #]& @
                          (* construct the body *)
                          Apply[F, Function[(* evaluated held body *)
                                   Hold[body]][Sequence @@ vars]]) /.
                                      function -> Function,
                                       F, Infinity, Heads -> True]]},
                  newBody /; (* are no other Slots present? *)
                  FreeQ[newBody, Slot, {-1}, Heads -> True]];
```

Here, the rule is applied twice to see successive replacement of the `Slots`.

```
#^2&[#]& /. rule[x]
```

```
% /. rule[y]
```

Applying now the rule until all pure functions are substituted gives our function `pureFunctionsWithSlotsTo·PureFunctionsWithVariables`.

```
pureFunctionsWithSlotsToPureFunctionsWithVariables[expr_] := expr //. rule[
```

Now, we apply `pureFunctionsWithSlotsToPureFunctionsWithVariables` to the example mentioned in the exercise.

```
f = #^2&[(#1 + #2)^3&[#1, 2#1]&[(#1 + #2 + (#^2&[#]))&[#1, #4]]]&
```

```
f1 = pureFunctionsWithSlotsToPureFunctionsWithVariables[f]
```

Here is a quick check that both two pure functions are mathematically identical.

```
{f[1, 2, 3, 4], f1[1, 2, 3, 4]}
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

18. Matrix Identities, Frobenius Formula, Iterative Matrix Square Root

a) We immediately have the following result, which shows that the relationship holds.

```
(#.#.# - Tr[#] #.# + 1/2(Tr[#]^2 - Tr[#.#]) # -
 Det[#] IdentityMatrix[3])&[Array[a, {3, 3}]] // Expand
```

Here is a similar identity for a 2×2 matrix:

$$\frac{2 - t\,\mathrm{tr}(\mathbf{A})}{\det(\mathbf{1} - t\,\mathbf{A})} = \sum_{k=0}^{\infty} \mathrm{tr}(\mathbf{A})\, t^k$$

```
Function[A, ((2 - t Tr[A])1/Det[IdentityMatrix[2] - t A]  -
    Sum[Evaluate[Tr[MatrixPower[A, n]]] t^n, {n, 0, Infinity}])][
                                        Array[a, {2, 2}]] // Simplify
```

And here is another form to express $\det(\mathbf{1} - t\,\mathbf{A})$ [211✱].

```
d = 2;
𝔸 = Table[𝕒[i, j], {i, d}, {j, d}];

Det[IdentityMatrix[d] - t 𝔸] // Simplify

Exp[-Sum[1/k t^k Tr[MatrixPower[𝔸, k]], {k, Infinity}]] // Simplify
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

b) We check the relationship explicitly.

```
A = Array[a, {2, 2}]; B = Array[b, {2, 2}];

B.A - (Tr[A.B] - Tr[A] Tr[B]) IdentityMatrix[2] -
       Tr[A] B - Tr[B] A + A.B // Expand
```

This relationship does not hold for 3×3 matrices.

```
A = Array[a, {3, 3}]; B = Array[b, {3, 3}];

(B.A - (Tr[A.B] - Tr[A] Tr[B]) IdentityMatrix[3] -
        Tr[A] B - Tr[B] A + A.B // Expand) ===
{{0, 0, 0}, {0, 0, 0}, {0, 0, 0}}
```

Now let us investigate if the generalized version exists. The function `makeSum` forms the inner sums in the proposed identity. The `c[i, j, k, l]` are to be determined unknowns.

```
makeSum[c_, {A_, B_}] :=
Sum[c[i, j, k, l] * Tr[MatrixPower[A, i].MatrixPower[B, j]] *
                      Tr[MatrixPower[A, k]] Tr[MatrixPower[B, l]],
    {i, 0, 1}, {j, 0, 1}, {k, 0, 1}, {l, 0, 1}]
```

To avoid calculations with large matrices that have symbolic entries, we generate now 15 pairs of "random" integer matrices and form the corresponding right-hand sides. If a solution for the `c[i, j, k, l]` exists, it must hold for these pairs too.

```
dim = 3;
eqs = Table[
A = Table[μ + 2 ν^3 + α, {μ, dim}, {ν, dim}];
B = Table[μ -ν^2 + α μ,  {μ, dim}, {ν, dim}];
# == 0& /@ Flatten[B.A -
     (makeSum[c[1], {A, B}] IdentityMatrix[dim] +
      makeSum[c[a, 1], {A, B}] A + makeSum[c[a, 2], {A, B}] A.A +
      makeSum[c[b, 1], {A, B}] B + makeSum[c[b, 2], {A, B}] B.B - A.B)],
           {α, 15}];
```

`eqs` contains many more equations than variables. For sufficiently generic pairs of matrices, we expect `eqs` either to yield a unique solution or no solution at all. We have more equations than unknowns.

```
cs = Cases[eqs, c[__][__], Infinity] // Union;

{Length[cs], Length[Flatten[eqs]]}
```

`Solve` shows that the system of equations for the `cs` is inconsistent. That means no identity of the above form holds for all 3 × 3 matrices.

```
Solve[Flatten[eqs], cs]
```

For similar identities, see [90✱].

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

c) Here are 2 n 3×3 matrices with generic symbolic elements.

```
n = 3;
Do[a[k] = Table[a[k, i, j], {i, n}, {j, n}], {k, 2n}]
```

There are the 720 possible permutations of the eight numbers {1, 2, 3, 4, 5, 6, 7, 8}.

```
perms = Permutations[Range[2 n]];
```

The proof now seems straightforward—we just evaluate an $n = 3$ version of the following input.

```
With[{n = 2},
 Block[{a, perms},
       Do[a[k] = Table[a[k, i, j], {i, n}, {j, n}], {k, 2 n}];
       (* the permutations *)
       perms = Permutations[Range[2 n]];
        Expand[Plus @@ (* the terms *)
                ((Signature[#] (Dot @@ (a /@ #)))& /@ perms)]]]
```

This process will theoretically work, but in practice, it will use a very large amount of memory. Let us see how large the quantities are and how long it takes to compute things. The calculation of a single matrix product is quite fast.

```
(m1 = Dot @@ (a /@ perms[[1]]);) // Timing
```

Every one of the 720 resulting matrix products needs about 0.5 MB in unexpanded and about 1 MB in expanded form.

```
{ByteCount[m1], ByteCount[m1 // Expand]}
```

So, we deal with each of the nine matrix elements individually. The next input will take about 3 minutes on a 2 GHz computer.

```
Table[sum = 0;
      Do[elem = (Dot @@ (a /@ perms[[k]]))[[i, j]];
         sum = sum + Signature[perms[[k]]] Expand[elem],
        {k, Length[perms]}];
      sum, {i, 3}, {j, 3}]
```

This method saved a lot of memory.

```
MaxMemoryUsed[]
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

d) We start by calculating the left-hand side of the eigenvalue equation for a generic degree n polynomial $q(x) = x^n + \sum_{i=0}^{n-1} \beta_i x^i$.

```
lhs[k_, n_] :=
Module[{p = x^k + Sum[α[j] x^j, {j, 0, k - 1}], ψ = Sum[β[j] x^j, {j, 0, n}
       Expand[D[p ψ, {x, k}]]]
```

The function coefficient extracts the coefficient of $\beta_l x^j$ of s. (Using the functions `Coefficient` and/or `Coeffi-cientList` the following could be implemented more efficiently; we will discuss these functions in Chapter 1 of the Symbolics volume [303✱].)

```
coefficient[s_, j_, l_] :=
Which[j == 0, s /. x -> 0,
      j == 1, (s /. x^_ -> 0 /. x -> C[1]) - (s /. x -> 0),
      True, (s /. x^j -> C[1]) - (s /. x -> 0)] /.
             x -> 0 /. β[l] -> C[2] /. _β -> 0 /. _C -> 1
```

For the calculation of the eigenvalues, we calculate the matrix of coefficients of $\beta_l x^j$ of `lhs[`*k*`, `*n*`]`.

```
cMat[k_, n_] := Table[coefficient[lhs[k, n], i, j], {i, 0, n}, {j, 0, n}]
```

Here are the first few $\lambda_j^{(n,k)}$.

```
Table[{k, Eigenvalues[cMat[k, 4]]}, {k, 0, 6}]
```

A quick look at the last numbers suggests $\lambda_j^{(n,k)} = (k+j)!/j!$.

```
Table[(n + k)!/n!, {k, 0, 6}, {n, 0, 4}]
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

e) For a shorter output, we define two format rules for the functions `dot` and `inverse`.

```
Format[dot[a__]] := Dot[a]
Format[inverse[a_]] := Power[a, "-1"]
```

These are the elementary properties of the dot product. We will denote the identity matrix (of unspecified dimension $n \times n$) by `one`. `a`, `b`, and `c` denote (sequences of) matrices.

```
(* flat like property *)
dot[a___, dot[b__], c___] :=  dot[a, b, c]
(* pull out numeric factors *)
dot[a___, f_?NumericQ b_, c___] :=  f dot[a, b, c]
(* single argument *)
dot[a_] := a
(* remove identities *)
dot[a___, b_, inverse[b_], c___] := dot[a, c]
dot[a___, inverse[b_], b_, c___] := dot[a, c]
dot[a___, one, b___] := dot[a, b]
dot[] := one
```

We add two more rules for dot. They are mathematically not needed, but they transform dot products into a nicer looking and more concise form.

```
(* partial expand *)
dot[a___, α_. one + b__, c___] := α dot[a, c] + dot[a, Plus[b], c]

(* extract minus sign *)
dot[a___, b_Plus, c___] := -dot[a, Expand[-b], c] /;
                Max[(List @@ b) /. {_dot -> 1, one -> 1}] < 0
```

Here is an example expression showing some of the now active transformation rules for `dot` at work.

```
Dot[A, Times[-2, Dot[-B, A, inverse[A], Dot[B, C]] + Dot[F, G]]]

% /. Dot -> dot
```

The internal form of the expression still contains `dot`.

```
InputForm[%]
```

For a pointed manipulation of dot products, we implement two more functions: `dotExpand` and `dotCollect`. `dotExpand[`*expr*`]` will expand all sums inside dot and `dotCollect[`*expr*`, `*v*`]` will collect terms in *expr* with respect to *v*.

```
dotExpand[expr_] := expr //.  dot[a___, b_ + c__, d___] :>
                               dot[a, b, d] + dot[a, Plus[c], d]

dotCollect[expr_, v_] := expr //.
{α_. dot[a___, v] + γ_. dot[c___, v] :> dot[α dot[a] + γ dot[c], v],
 α_. dot[v, b___] + γ_. dot[v, c___] :> dot[v, α dot[b] + γ dot[c]],
 α_. dot[a___, v] + β_. v :> dot[α dot[a] + β one, v],
 α_. dot[v, b___] + β_. v :> dot[v, α dot[b] + β one]}
```

Given two expressions of the form *term* = *rest*, `isolate[`*term*`, `*rest*`, `*v*`]` will multiply `rest` with appropriate inverses to isolate the variable *v* in *term*.

```
isolate[f_?NumericQ term_, rest_, v_] :=
        isolate[term, dot[rest, f], v]

isolate[dot[a_, b___, v_, c___], rest_, v_] :=
        isolate[dot[b, v, c], dot[inverse[a], rest], v]

isolate[dot[a___, v_, b___, c_], rest_, v_] :=
        isolate[dot[a, v, b], dot[rest, inverse[c]], v]

isolate[v_, rest_, v_]  := {v, rest}
```

Here is an example.

```
isolate[dot[a, A, b], C, A]
```

Using the function `isolate`, it is straightforward to implement a function `solve[`*eqs*`, `*v*`]` that solves *eqs* = 0 for *v*.

```
solve[eqs_, v_] :=
Module[{eqsC = dotCollect[eqs, v], bTerm},
  (* the term that contains v *)
  bTerm = Select[dotCollect[eqs, v], MemberQ[#, v, {0, Infinity}]&];
  (* return result as a rule *)
  Rule @@ isolate[bTerm, bTerm - eqsC, v]]
```

The following input shows `solve` at work.

```
solve[dot[𝔸, 𝕓] - 2 dot[𝔸', 𝕓] - dot[𝔹, 𝕖] + dot[ℂ, 𝕙], 𝕓]
```

Using `solve`, we now implement the function `reduce`. `reduce[`*eqs*`, `*n*`, `*v*`]` eliminates the variable *v* from the equations *eqs* using the *n*th equation of *eqs*.

```
reduce[eqs_, n_, v_] := Delete[eqs, n] //. solve[eqs[[n]], v]
```

Now let us put the functions `solve` and `reduce` to work. Let $\begin{pmatrix} \mathbb{A} & \mathbb{B} \\ \mathbb{C} & \mathbb{D} \end{pmatrix}$ be a block matrix and $\begin{pmatrix} 𝕒 & 𝕓 \\ 𝕔 & 𝕕 \end{pmatrix}$ its inverse. This means we have the following set of coupled, linear equations for 𝕒, 𝕓, 𝕔, and 𝕕.

```
(eqs0 = Inner[dot, {{𝔸, 𝔹}, {ℂ, 𝔻}}, {{𝕒, 𝕓}, {𝕔, 𝕕}}, Plus] -
              {{𝕠𝕟𝕖, 0}, {0, 𝕠𝕟𝕖}} // Flatten) // TableForm
```

It is straightforward to solve, say, for 𝕕 (depending on the properties of the various block matrices other orders might be more appropriate [191✱]). We eliminate 𝕓, 𝕒, and 𝕔 and solve the remaining single equation for 𝕕.

```
eqs1 = reduce[eqs0, 2, 𝕓]

eqs2 = reduce[eqs1, 2, 𝕒]

eqs3 = reduce[eqs2, 1, 𝕔]

sol𝕕 = solve[eqs3[[1]], 𝕕]
```

Now, we have two possibilities to solve for the remaining variables 𝕒, 𝕓, and 𝕔. Either we repeat the above steps for a different variable ordering. Or we backsubstitute the solution for 𝕕 into `eqs3` and obtain so the solution for 𝕔 and then backsubstitute the solutions for 𝕔 and 𝕕 into `eqs2` to obtain the solution for 𝕒 and so on. To simplify intermediate expressions that arise after backsubstitution, we implement a very simplistic simplifier. `dotSimplify` is basically equal to `dotCollect`, but this time we do not prescribe *v*.

```
dotSimplify[expr_] := expr //.
{α_. dot[a___, v_] + γ_. dot[c___, v_] :> dot[α dot[a] + γ dot[c], v],
 α_. dot[v_, b___] + γ_. dot[v_, c___] :> dot[v, α dot[b] + γ dot[c]],
 α_. dot[a___, v_] + β_. v_ :> dot[α dot[a] + β 𝕠𝕟𝕖, v],
 α_. dot[v_, b___] + β_. v_ :> dot[v, α dot[b] + β 𝕠𝕟𝕖]}
```

Substituting the solution for 𝕕 into eqs2 gives one equation for 𝕔.

```
eqs2a = dotSimplify[eqs2 /. sol𝕕]

sol𝕔 = solve[eqs2a[[1]], 𝕔]
```

Substituting the solutions for 𝕔 and 𝕕 into eqs1 gives two equations for 𝕒. Both can be used to solve for 𝕒.

```
eqs1a = dotSimplify[eqs1 /. sol𝕕 /. sol𝕔]

sol𝕒1 = solve[eqs1a[[1]], 𝕒]

sol𝕒2 = solve[eqs1a[[2]], 𝕒]
```

The identity of the two forms can be easily established by multiplying by the inverse plus term.

```
dot[sol𝕒1[[2]] - sol𝕒2[[2]],
    𝔹 - dot[𝔸, inverse[ℂ], 𝔻]] // dotExpand
```

Substituting finally the solutions for 𝕔, 𝕕, and 𝕒 into eqs0 gives three equations for 𝕓. All three can be used to solve for 𝕓.

```
eqs0a = dotSimplify[eqs0 /. sol𝕕 /. sol𝕔 /. sol𝕒2]

sol𝕓 = solve[eqs0a[[2]], 𝕓]
```

So, we obtain the following result for the inverse of a 2×2 block matrix.

```
res = {{𝕒, 𝕓}, {𝕔, 𝕕}} /. sol𝕒1 /. sol𝕓 /. sol𝕔 /. sol𝕕
```

Here is a quick check of the result. We use the function BlockMatrix from the package LinearAlgebra`MatrixManipulation` to assemble a 2×2 block matrix, each submatrix is a generic 2×2 matrix with symbolic entries. (We could, of course, use larger matrices here.)

```
Needs["LinearAlgebra`MatrixManipulation`"]

{𝔸1, 𝔹1, ℂ1, 𝔻1} =
Table[Subscript[#, i, j], {i, 2}, {j, 2}]& /@ {a, b, c, d};

𝔸𝔹ℂ𝔻 = BlockMatrix[{{𝔸1, 𝔹1}, {ℂ1, 𝔻1}}]

Simplify @ (Inverse[𝔸𝔹ℂ𝔻] -
Block[{𝕠𝕟𝕖 = {{1, 0}, {0, 1}},
       𝔸 = 𝔸1, 𝔹 = 𝔹1, ℂ = ℂ1, 𝔻 = 𝔻1},
       Evaluate[BlockMatrix[res /. dot -> Dot /. inverse -> Inverse]]])
```

For a 3×3 matrix, we can repeat all of the above steps. We solve the system for 𝕕.

```
eqs0 = Inner[dot, {{𝔸, 𝔹, ℂ}, {𝔻, 𝔼, 𝔽}, {𝔾, ℍ, 𝕀}},
                  {{𝕒, 𝕓, 𝕔}, {𝕕, 𝕖, 𝕗}, {𝕘, 𝕙, 𝕚}}, Plus] -
       {{𝕠𝕟𝕖, 0, 0}, {0, 𝕠𝕟𝕖, 0}, {0, 0, 𝕠𝕟𝕖}} // Flatten

(* recursively eliminate variables *)
eqs1 = dotExpand /@ reduce[eqs0, 2, 𝕓];
eqs2 = dotExpand /@ reduce[eqs1, 2, 𝕔];
eqs3 = dotExpand /@ reduce[eqs2, 2, 𝕒];
eqs4 = dotExpand /@ reduce[eqs3, 3, 𝕗];
eqs5 = dotExpand /@ reduce[eqs4, 3, 𝕘];
eqs6 = dotExpand /@ reduce[eqs5, 3, 𝕙];
eqs7 = dotExpand /@ reduce[eqs6, 3, 𝕚];
eqs8 = dotExpand /@ reduce[eqs7, 2, 𝕖];
sol𝕕 = solve[eqs8[[1]], 𝕕]
```

Here is again a quick check for the correctness of the last result. To avoid the symbolic inversion of a 6×6 matrix, we use a matrix with numeric elements here.

```
{𝔸1, 𝔹1, ℂ1, 𝔻1, 𝔼1, 𝔽1, 𝔾1, ℍ1, 𝕀1} =
       (* use some rational function of the indices *)
       Table[Table[k/(i + j + k + 1), {i, 2}, {j, 2}], {k, 9}];

𝔸𝔹ℂ𝔼𝔽𝔾ℍ𝕀𝕁 =
BlockMatrix[{{𝔸1, 𝔹1, ℂ1}, {𝔻1, 𝔼1, 𝔽1}, {𝔾1, ℍ1, 𝕀1}}];

Take[Inverse[𝔸𝔹ℂ𝔼𝔽𝔾ℍ𝕀𝕁], {3, 4}, {1, 2}]

Block[{𝔸 = 𝔸1, 𝔹 = 𝔹1, ℂ = ℂ1, 𝔻 = 𝔻1, 𝔼 = 𝔼1,
       𝔽 = 𝔽1, 𝔾 = 𝔾1, ℍ = ℍ1, 𝕀 = 𝕀1},
       Evaluate[sol𝕕 /. dot -> Dot /. inverse -> Inverse]]
```

Instead of solving manually for the remaining eight block matrices 𝕒, 𝕓, 𝕔, 𝕖, 𝕗, 𝕘, 𝕙, and 𝕚, we implement a function `fullSolve` that carries out these steps.

```
fullSolve[eqs_List, elimVars_, v_] :=
Module[{remainingEqs = eqs,
        remainingElimVars = Alternatives @@ elimVars,
        𝕠𝕟𝕖FreeEquations, nEq, nextElimVar, nextEq},
 While[Length[remainingEqs] > 1,
       (* use first equations without identity *)
       𝕠𝕟𝕖FreeEquations = Select[remainingEqs,
                                FreeQ[#, 𝕠𝕟𝕖, Infinity]&];
       If[𝕠𝕟𝕖FreeEquations =!= {},
          nEq = Position[remainingEqs,
                         𝕠𝕟𝕖FreeEquations[[1]]][[1, 1]];
          (* variable to be eliminated *)
          nextElimVar = Cases[𝕠𝕟𝕖FreeEquations[[1]],
                              remainingElimVars, Infinity][[1]],
          (* equation to be used *)
          nextEq = Select[remainingEqs,
                          MemberQ[#, remainingElimVars,
                                       Infinity]&, 1][[1]];
          nEq = Position[remainingEqs, nextEq][[1, 1]];
          nextElimVar = Cases[nextEq, remainingElimVars, Infinity][[1]]];
        (* eliminate one variable *)
        remainingEqs = dotExpand /@ reduce[remainingEqs, nEq, nextElimVar];
        remainingElimVars = DeleteCases[remainingElimVars, nextElimVar]];
(* solve remaining equation *)
solve[remainingEqs[[1]], v]]
```

`fullSolve` allows to rederive the above solution for 𝕕 as well as to calculate the other inverses.

```
fullSolve[eqs0, {𝕒, 𝕓, 𝕔, 𝕖, 𝕗, 𝕘, 𝕙, 𝕚}, 𝕕]

fullSolve[eqs0, {𝕒, 𝕓, 𝕔, 𝕕, 𝕖, 𝕗, 𝕘, 𝕙}, 𝕚]

{{𝕒, 𝕓, 𝕔}, {𝕕, 𝕖, 𝕗}, {𝕘, 𝕙, 𝕚}} /. Table[
fullSolve[eqs0, Delete[{𝕒, 𝕓, 𝕔, 𝕕, 𝕖, 𝕗, 𝕘, 𝕙, 𝕚}, k],
                      {𝕒, 𝕓, 𝕔, 𝕕, 𝕖, 𝕗, 𝕘, 𝕙, 𝕚}[[k]]],
          {k, 9}]
```

A quick check for the derived result.

```
Inverse[𝔸𝔹ℂ𝔼𝔽𝔾ℍ𝕀𝕁] - BlockMatrix @
Block[{𝔸 = 𝔸1, 𝔹 = 𝔹1, ℂ = ℂ1, 𝔻 = 𝔻1, 𝔼 = 𝔼1,
       𝔽 = 𝔽1, 𝔾 = 𝔾1, ℍ = ℍ1, 𝕀 = 𝕀1},
       Evaluate[% /. dot -> Dot /. inverse -> Inverse]]
```

Now let us deal with the case of a 4×4 block matrix. This is the defining set of equations for the 16 inverse matrices

$a_{i,j}$.

```
m = 4;
eqs0 = Inner[dot, Table[Subscript[A, i, j], {i, m}, {j, m}],
                  Table[Subscript[a, i, j], {i, m}, {j, m}], Plus] -
       DiagonalMatrix[Table[one, {m}]] // Flatten
```

For brevity, we solve only for $a_{1,1}$. We use the above-implemented function `fullSolve` for the solution (instead of, say, iterate the 2×2 result).

```
allVars = Flatten[Table[Subscript[a, i, j], {i, m}, {j, m}]]
```

```
ν = 1;
a11 = fullSolve[eqs0, Delete[allVars, ν], allVars[[ν]]];
```

The result is quite large. To represent it in a compact form, we repeatedly introduce some abbreviations for inverses of sums.

```
a11 // LeafCount
```

```
invAbb1 = Cases[a11, inverse[_Plus], Infinity, 1]
```

```
a11Short1 = a11 //. invAbb1[[1]] -> A;
```

```
invAbb2 = Cases[a11Short1, inverse[_Plus], Infinity, 1]
```

```
a11Short2 = a11Short1 //. invAbb2[[1]] -> B;
```

Here is the simplified form of the result.

```
dotSimplify[a11Short2]
```

For the solution of more complicated blockmatrix problems, see [177*], [317*], and http://math.ucsd.edu/~ncalg/ and L. Zhao's *MathSource* package 0212-016. For supermatrices, see [22*] and [87*].

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

f) The function `MatrixSquareRoot` implements the iterative procedure.

```
MatrixSquareRoot[A_?(MatrixQ[#, NumericQ]&), maxIter_:100] :=
  FixedPointList[(#.(#.# + 3 A).Inverse[3 #.# + A])&,
                 IdentityMatrix[Length[A]], maxIter]
```

This is the Hilbert matrix whose square root has to be found.

```
h = Table[1/(i + j + 1), {i, 10}, {j, 10}];
```

A machine-precision calculation does not converge. The fifth iteration yields a result correct to about five digits. Further iterations diverge. The message `Inverse::luc` indicates that after a certain number of iterations (eight) the inverse cannot be calculated reliably anymore.

```
msrMP = MatrixSquareRoot[N[h]];
```

```
{msrMP // Length, Max[Abs[#.# - h]]& /@ Take[msrMP, 10]}
```

A high-precision calculation starting with 400 digits yields a square root having about 200 correct digits.

```
msrHP = MatrixSquareRoot[N[h, 400]];
```

```
{(* iteration data *) Length[msrHP], Precision[msrHP[[-1]]],
 (* check result *) msrHP[[-1]].msrHP[[-1]] - h // Abs // Max,
 1 - msrHP[[-1]]/MatrixPower[N[h, 1000], 1/2] // Abs // Max}
```

For other iterative methods to calculate the square root of a matrix, see [136*], [109*].

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

g) We start by implementing the three determinants det($\mathbf{G}(a, b)$), det($\mathbf{W}(x)$), and det($\mathbf{M}(x_1, \ldots, x_n)$). Here *fs* is a list of functions $\{f_1, \ldots, f_n\}$.

```
GramDet[fs_List, {a_, b_}] := Det @
Outer[Integrate[#1 #2, {ξ, a, b}]&, #[ξ]& /@ fs, #[ξ]& /@ fs]

WronskiDet[fs_List, x_] := Det @
Table[D[#[x]& /@ fs, {x, k}], {k, 0, Length[fs] - 1}]

FunDet[fs_List, xs_List] := Det @ Outer[#1[#2]&, fs, xs]
```

In the following, we will always use the standard variables and so define the following three shortcuts.

```
GramDet[n_Integer] := GramDet[Array[f, n], {a, b}]
WronskiDet[n_Integer] := WronskiDet[Array[f, n], x]
FunDet[n_Integer] := FunDet[Array[f, n], Array[x, n]]
```

We start with the first identity.

```
𝕗[n_] := Product[k^(n - Abs[n - k]), {k, 2n - 1}]/(n^2)!

Table[Timing[Expand[WronskiDet[n]^2 /. x -> a] -
             Expand[𝕗[n] D[GramDet[n], {b, n^2}] /. b -> a]],
      {n, 1, 4}]
```

We see dramatic increase in the calculation time as a function of n. The time-consuming operations are the n^2 differentiations with respect to b. Actually, *Mathematica* has optimized code for higher order differentiations. Carrying out the differentiations repeatedly takes considerably longer. Here is a list of the timing and the number of terms in the intermediate sums.

```
Module[{gd = GramDet[4]},
       Table[{j, Timing[gd = D[gd, b]][[1]], Length[gd]}, {j, 16}]]
```

Waiting long enough, we could also prove the $n = 5$ case explicitly.

Now, we will deal with the second identity. A straightforward implementation does not yield a verifiable identity.

```
With[{n = 2},
     Integrate[FunDet[n]^2, {x[1], a, b}, {x[2], a, b}] -
     GramDet[n]] // ExpandAll
```

`Integrate` does not automatically distribute over sums (because a sum might be integrable in closed form, but its individual summands might not be). So, we distribute `Integrate` over sums and rename the dummy integration variables afterwards. This allows to verify the identities for $n = 1, 2, 3, 4, 5$ easily. Because no feature of the built-in function `Integrate` are used in the following calculation, but only structural operations based on the form of the expressions are carried out, we use `Block` with the local variable `Integrate`. This avoids that the built-in function `Integrate` tries to integrate the expressions, and is so much faster.

```
Block[{Integrate},
Table[Timing[Expand[
     (Expand[1/n! Fold[Integrate[#1, {#2, a, b}]&,
                Expand[FunDet[n]^2], Table[x[j], {j, n}]] //.
     (* distribute Integrate over sums *)
     Integrate[p_Plus, {ξ_, a_, b_}] :>
               (Integrate[#, {ξ, a, b}]& /@ p)] //.
     (* pull-out integration variable free factors *)
     Integrate[f_ g_, {ξ_, a_, b_}] :>
       f Integrate[g, {ξ, a, b}] /; FreeQ[f, ξ, {0, Infinity}] //.
     (* use ξ for dummy integration variables *)
     Integrate[int_, {x[j_], a, b}] :>
               Integrate[int /. x[j] -> ξ, {ξ, a, b}]) -
     GramDet[n]]], {n, 2, 5}]]
```

We can improve on the last timing by observing that the built-in function `Integrate` does a lot of work to find matching internal integration rules. By implementing our own function `integrate` that is linear and pulls out integration variable-independent factors, we can deal with the $n = 4$ case, and $n = 5$ case too.

```
(* linearity of integration *)
integrate[p_Plus, {x_, a, b}] := integrate[#, {x, a, b}]& /@ p;

integrate[c_ f_, {x_, a, b}] := c integrate[f, {x, a, b}] /;
                                          FreeQ[c, x, {0, Infinity}];

Table[Timing[Expand[
(Expand[1/n! Fold[integrate[#1, {#2, a, b}]&,
                Expand[FunDet[n]^2], Table[x[j], {j, n}]] //.
    integrate[int_, {x[j_], a, b}] :>
              integrate[int /. x[j] -> ξ, {ξ, a, b}]] /.
    integrate -> Integrate) - GramDet[n]]], {n, 4, 5}]
```

The third identity is most easily verifiable. Because the number of terms does not grow after differentiation, this time we can easily reach $n = 8$.

```
Table[{n, Timing[Expand[(Fold[D, FunDet[n],
                         Table[{x[j], j - 1}, {j, 2, n}]] /.
                    x[_] :> x) - WronskiDet[n]]]}, {n, 8}]
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

h) This is the indexed version of the definition.

```
d0[fA_?MatrixQ, A_?MatrixQ] :=
  With[{d = Length[A]},
       Table[D[fA[[i, j]], A[[k, l]]], {i, d}, {j, d}, {k, d}, {l, d}]]
```

Here is a possible index-free version of the definition.

```
d1[fA_?MatrixQ, A_?MatrixQ] :=
   Map[Function[e, Map[D[e, #]&, A, {2}]], fA, {2}]
```

Using the function `Outer`, we can further shorten the definition.

```
d2[fA_?MatrixQ, A_?MatrixQ] := Outer[D, fA, A]
```

Here is a quick check that the three definitions are identical for small matrix dimensions and powers.

```
Table[A = Table[a[i, j], {i, d}, {j, d}];
      fA = MatrixPower[A, n];
      SameQ @@ Expand[{d0[fA, A], d1[fA, A], d2[fA, A]}], {d, 4}, {n, 4}]
```

We continue with the implementation of the special formula for the derivative of a positive integer power of a matrix.

Here is again the straightforward index-using implementation of the definition.

```
dP0[An_?MatrixQ, A_?MatrixQ] :=
    With[{d = Length[A]},
         Table[Sum[MatrixPower[A, m - 1][[i, k]]*
                   MatrixPower[A, n - m][[l, j]], {m, n}],
                    {i, d}, {j, d}, {k, d}, {l, d}]]
```

And here is an index-free implementation. We have to carry out a nontrivial transposition on the outer products to obtain the correct index ordering.

```
dP2[An_?MatrixQ, A_?MatrixQ] := Transpose[#, {4, 2, 1, 3}]& @
    Sum[Outer[Times, MatrixPower[A, m - 1], MatrixPower[A, n - m]], {m, n}]
```

And here is a again quick check that the two definitions are identical.

```
Table[A = Table[a[i, j], {i, d}, {j, d}];
      An = MatrixPower[A, n];
      SameQ @@ Expand[{dP0[An, A], dP2[An, A]}], {d, 4}, {n, 4}]
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

19. Autoloading and Package Test

a) These are all built-in function names.

```
allNames = Names["*"];
```

This is the amount of memory currently used by *Mathematica*.

```
MemoryInUse[]
```

We determine all definitions currently present for all of them.

```
(* string to unevaluated expression *)
unevaluatedNamesInitially =
          ToExpression[#, InputForm, Unevaluated]& /@ allNames;
```

We make all definitions available by removing the `ReadProtected` attribute.

```
readProtectedNames =
Select[unevaluatedNamesInitially, MemberQ[Attributes[#], ReadProtected]&];
```

About 200 functions of this kind exist.

```
Length[readProtectedNames]
```

```
Off[Attributes::"locked"];
ClearAttributes[#, ReadProtected]& /@ readProtectedNames;
```

These are all current definitions. Because the symbol `I` is has the `Locked` and the `ReadProtected` attribute, we turn off the `General::readp` message.

```
Off[General::"readp"];
(* all currently known rules *)
OwnValuesInitially    = OwnValues    /@ unevaluatedNamesInitially;
DownValuesInitially   = DownValues   /@ unevaluatedNamesInitially;
NValuesInitially      = NValues      /@ unevaluatedNamesInitially;
FormatValuesInitially = FormatValues /@ unevaluatedNamesInitially;
SubValuesInitially    = SubValues    /@ unevaluatedNamesInitially;
UpValuesInitially     = UpValues     /@ unevaluatedNamesInitially;
```

Most present are `OwnValues`.

```
Count[#, _?(# =!= {}&)]& /@
  {OwnValuesInitially, DownValuesInitially, NValuesInitially,
   FormatValuesInitially, SubValuesInitially, UpValuesInitially}
```

It is the ownvalue that causes autoloading of the start-up packages. Here is the current ownvalue of `AppellF1` (a special function of mathematical physics) shown.

```
OwnValues[AppellF1]
```

Currently, no downvalues are associated with `AppellF1`.

```
DownValues[AppellF1]
```

Evaluating the symbol itself causes the right-hand side of the last rule to evaluate, and as a result, the corresponding *Mathematica* package gets loaded.

```
AppellF1
```

As a result, no ownvalues exist anymore for `AppellF1`.

```
OwnValues[AppellF1]
```

But the loading of the package did create downvalues for `AppellF1`.

```
Begin["System`AppellF1Dump`"]
DownValues[AppellF1]
End[]
```

If we want to determine which symbols are autoloaded, we have to watch for the head `System`Dump`AutoLoad` at position `{1, 2, 1, 0}` in the corresponding ownvalues. Here, this head is extracted for `InverseJacobiCD`, another special function of mathematical physics.

```
OwnValues[InverseJacobiCD][[1, 2, 1, 0]]
```

Going systematically through all function names yields the following list of autoloaded functions. Most autoloaded functions are special functions of mathematical physics (see Chapter 3 of the Symbolics volume [303✱]).

```
Off[Part::"partd"]; Off[Part::"partw"];
First /@ Select[Transpose[{allNames, OwnValuesInitially}],
       (If[Depth[#[[2]]] > 3,
          #[[2, 1, 2, 1, 0]]] === System`Dump`AutoLoad)&]

Length[%]
```

Converting all names into expressions and evaluating them yields fewer functions with ownvalues (because the autoload· ing ownvalues were removed), but more functions with downvalues. (Because the autoloading results in reading in definitions for these functions.)

```
ToExpression /@ allNames;

(* removing again the ReadProtected attribute;
   in the process of loading the package it might have been added *)
readProtectedNames = Select[unevaluatedNamesInitially,
                            MemberQ[Attributes[#], ReadProtected]&];

Off[General::"readp"];
OwnValuesAfter    = OwnValues    /@ unevaluatedNamesInitially;
DownValuesAfter   = DownValues   /@ unevaluatedNamesInitially;
NValuesAfter      = NValues      /@ unevaluatedNamesInitially;
FormatValuesAfter = FormatValues /@ unevaluatedNamesInitially;
SubValuesAfter    = SubValues    /@ unevaluatedNamesInitially;
UpValuesAfter     = UpValues     /@ unevaluatedNamesInitially;
```

The most present values are ownvalues.

```
Count[#, _?(# =!= {}&)]& /@
  {OwnValuesAfter, DownValuesAfter, NValuesAfter,
   FormatValuesAfter, SubValuesAfter, UpValuesAfter}
```

This is the amount of memory used after all autoloaded functions have their full definitions. Now, *Mathematica* uses much more memory than in the beginning of this session.

```
MemoryInUse[]
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

b) Here are all packages to be investigated.

```
Length @ (files = Flatten[
 FileNames["*.m", #, Infinity]& /@
       Select[$Path, StringMatchQ[#, "*StandardPackages*"]&]])
```

To avoid the multiple appearance of commands as much as possible (we cannot avoid them completely because a couple of packages need the same ones), we eliminate all master packages from `allFiles`. (We cannot avoid all multiple appearances without a much larger effort; the evaluation of `Needs` inside packages causes some problems). Because of the intricate way the univariate and the multivariate statistics packages work together, we also do not take them into account here.

```
files = Complement[files,
   Select[files, (StringMatchQ[#, "*Master*"] ||
                  StringMatchQ[#, "*Kernel*"] ||
                  StringMatchQ[#, "*Common*"] ||
                  StringMatchQ[#, "*Statistics*"])&]];
```

Here are the names of the variables introduced, which will be used in the following. To get them in the list of symbols before any package is loaded, we introduce them now.

```
(* introduce all symbols *)
namesBefore; allNamesBefore; unevaluatedNamesBefore;
attributesBefore; optionsBefore; $ContextPathBefore;
allPackageVariables; exportedPackageCommands;
exportedDocumentedPackageCommands;
exportedUndocumentedPackageCommands;
messageGeneratingPackages; attributesChangingPackages;
optionsChangingPackages; exportedUndocumentedCommands;
namesAfter; newNames; allNamesAfter; allNewNames;
documentedCommands; attributesAfter; posis; optionsAfter; i;
commonAttributeChanges; commonExportedDocumentedPackageCommands;
commonExportedPackageCommands; commonExportedUndocumentedPackageCommands;
commonOptionChanges; exportedDocumentedPackageCommands1;
exportedPackageCommands1; exportedUndocumentedPackageCommands1;
reducedFileName;
```

For comparison with the condition after reading in the packages, here are the known variable names (collected as strings), their attributes, and their options.

```
(* the function names *)
namesBefore = DeleteCases[Names["*"], "$Echo"];
(* evaluate all function names to avoid auto-loading later on *)
ToExpression /@ namesBefore;
(* all names from all contexts *)
allNamesBefore = Names["*`*"];
(* transform strings into unevaluated commands *)
unevaluatedNamesBefore =
 ToExpression[#, InputForm, Unevaluated]& /@ namesBefore;
(* list of current attributes *)
attributesBefore = Attributes /@ unevaluatedNamesBefore;
(* list of current options *)
optionsBefore = Options /@ unevaluatedNamesBefore;
(* the original context path *)
$ContextPathBefore = $ContextPath;

{Length[allNamesBefore], Length[namesBefore], MemoryInUse[]}
```

Here is the list that will collect all symbols.

```
allPackageVariables = {};
```

This list collects all exported symbols.

```
exportedPackageCommands = {};
```

This list collects all exported and documented symbols. (In the ideal case, this list should be identical to the list (`exportedPackageCommands`.)

```
exportedDocumentedPackageCommands = {};
```

This list collects all exported but undocumented commands. (If possible, this list should be empty.)

```
exportedUndocumentedPackageCommands = {};
```

This one collects all packages that generate messages. (This will be mainly the case for obsolete packages.)

```
messageGeneratingPackages = {};
```

The following collects all packages that change attributes of built-in functions from `namesBefore`.

```
attributesChangingPackages = {};
```

This collects all packages that change options of built-in functions from `namesBefore`.

```
optionsChangingPackages = {};
```

Now, we turn to the real work, the analysis of the loading and contents of all packages. Taking into account the naming of variables that appear in the following, together with the code comments, the operation of the following code should be obvious. After the analysis of the loading process and the symbols used, the new symbols are removed using `Remove`. For a more refined treatment of restoring the state of a *Mathematica* session, see the package `CleanSlate` by T. Gayley (*MathSource* 0204-310).

We will get some messages that originate from loading obsolete packages, but because some of the symbols exported are not present in the `System`` context, we cannot shut off these messages now.

```
Off[SetOptions::optnf]; Off[StringJoin::string]; Off[MessageName::messg];

Do[
  (* read in file and check if this generates a message *)
  Check[Get[files[[i]]], AppendTo[messageGeneratingPackages, i]];
  (* analyze all that could have been changed,
    and save changes in corresponding lists;
    after that, restore original state *)
 (* remove disturbing  definitions *)
 Unset[$Pre]; Unset[$Post];
 (* new names globally visible *)
  namesAfter = Names["*"];
  newNames = Complement[namesAfter, namesBefore];
  AppendTo[exportedPackageCommands, newNames];
  (* new names from all contexts *)
  allNamesAfter = Names["*`*"];
  allNewNames = Complement[allNamesAfter, allNamesBefore];
  AppendTo[allPackageVariables, allNewNames];
(* exported and documented commands *)
documentedCommands = ToString /@ Select[
       ToExpression[#, InputForm, Unevaluated]& /@ newNames,
               Head[MessageName[#, "usage"]] == String&];
AppendTo[exportedDocumentedPackageCommands, documentedCommands];
AppendTo[exportedUndocumentedPackageCommands,
        Complement[newNames, documentedCommands]];
(* checking the status of the attributes *)
attributesAfter = Attributes /@ unevaluatedNamesBefore;
If[attributesAfter =!= attributesBefore,
   posis = Flatten[Position[Apply[SameQ,
        Transpose[{attributesAfter, attributesBefore}], {1}], False]];
   AppendTo[attributesChangingPackages, {i, posis}];];
(* checking the status of the options *)
optionsAfter = Options /@ unevaluatedNamesBefore;
If[optionsAfter =!= optionsBefore,
   posis = Flatten[Position[Apply[SameQ,
        Transpose[{optionsAfter, optionsBefore}], {1}], False]];
   AppendTo[optionsChangingPackages, {i, posis}];
   Unprotect /@ namesBefore[[posis]];
   Do[Options[namesBefore[[posis[[i]]]]] = optionsBefore[[i]],
     {i, Length[posis]}]];
(* restoring old state *)
   Do[If[FreeQ[attributesBefore[[i]], Locked],
         Attributes[Evaluate[namesBefore[[posis[[i]]]]]] =
                    attributesBefore[[i]]],
     {i, Length[posis]}];
(* remove introduced variables *)
Unprotect /@ allNewNames;
(* some screened symbols will be removed automatically *)
Off[Remove::rmnsm]; Off[Remove::relex];
Remove /@ allNewNames;
On[Remove::rmnsm]; On[Remove::relex];
(* restore old context path *)
$ContextPath = $ContextPathBefore, {i, Length[files]}]
```

We will remove some commonly appearing commands that are related to the front end in the following input.

```
commonExportedPackageCommands =
First /@ Select[Split[Sort[Flatten[exportedPackageCommands]]],
       Length[#] > 5&]
```

```
exportedPackageCommands1 =
  Complement[#, commonExportedPackageCommands]& /@
                              exportedPackageCommands;
```

Here is a list of how many packages export how many commands. Some packages seem to export many functions. This is because some packages load other packages recursively.

```
{#[[1]], Length[#]}& /@
  Split[Sort[Length /@ exportedPackageCommands1]]
```

This makes a total of about 2000 different exported commands.

```
Length[Union[Flatten[exportedPackageCommands1]]]
```

Now, let us look at the documented commands.

```
commonExportedDocumentedPackageCommands =
First /@ Select[Split[Sort[Flatten[exportedDocumentedPackageCommands]]],
       Length[#] > 5&];
```

```
exportedDocumentedPackageCommands1 =
        Complement[#, commonExportedDocumentedPackageCommands]& /@
                              exportedDocumentedPackageCommands;
```

```
{#[[1]], Length[#]}& /@
  Split[Sort[Length /@ exportedDocumentedPackageCommands1]]
```

Now, let us look at the undocumented, but exported commands.

```
commonExportedUndocumentedPackageCommands =
First /@ Select[Split[Sort[Flatten[
        exportedUndocumentedPackageCommands]]], Length[#] > 5&];
```

```
(exportedUndocumentedPackageCommands1 =
 Union[Flatten[Complement[#,
    commonExportedUndocumentedPackageCommands]& /@
                exportedUndocumentedPackageCommands]]) // Length
```

Which packages generated messages while loading them? Most of these packages deal with obsolete functions.

```
reducedFileName =
     StringDrop[#, {1, StringPosition[#, "StandardPackages"][[1, 2]]}]&;
```

```
reducedFileName /@ files[[messageGeneratingPackages]]
```

Which packages changed the attributes of built-in functions? And which functions were changed?

```
commonAttributeChanges =
First /@ Select[Split[Sort[Flatten[attributesChangingPackages]]],
       Length[#] > 5&];
```

```
{reducedFileName[files[[#[[1]]]]], namesBefore[[#[[2]]]]}& /@
DeleteCases[{#[[1]], Complement[#[[2]], commonAttributeChanges]}& /@
                                attributesChangingPackages, {_, {}}]
```

Which packages changed the options of built-in functions? And which functions were changed?

```
commonOptionChanges =
First /@ Select[Split[Sort[Flatten[optionsChangingPackages]]],
       Length[#] > 5&];
```

```
{reducedFileName[files[[#[[1]]]]], namesBefore[[#[[2]]]]}& /@
DeleteCases[{#[[1]], Complement[#[[2]], commonOptionChanges]}& /@
                                 optionsChangingPackages, {_, {}}]
```

Here is the state of *Mathematica* after loading all packages.

```
{Length[Names["*`*"]], Length[Names["*"]], MemoryInUse[]}
```

Considering that, we have removed all definitions that were read in immediately, we lost some memory. Using `Share`, we can recover some of it.

```
Share[]
```

The following number of variables have been added since the beginning.

```
Complement[Names["*`*"], allNamesBefore]
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

20. `PrecedenceForm`

The command `PrecedenceForm` determines the bracketing in using the infix notation for commands. Here is the syntax: *command*`[`*argument*$_1$`, ..., PrecedenceForm[`*argument*`], ` *precedenceLevel*`], ..., ` *argument*$_n$`]` specify precedence of argument for formatting. Here, *precedenceLevel* must be a positive integer. The result is then printed with appropriate parentheses if the command would have the precedence *precedenceLevel*. Here is an example.

```
Plus[x, PrecedenceForm[y, 100], z]
```

```
Plus[x, PrecedenceForm[y, 500], z]
```

We begin with the search for all commands where a sensible `PrecedenceForm` could exist. We do this by checking to see whether a round pair of brackets `()` appears in *command*`[x, PrecedenceForm[y, 1]]`. Because many built-in commands will not be happy to get this input, we first turn off all messages and remove commands that are especially dangerous for our investigations.

```
(* all built-in names *)
systemCommands = Names["System`*"];

(* read in message file *)
Get[ToFileName[{$TopDirectory, "SystemFiles", "Kernel",
                "TextResources", $Language}, "Messages.m"]];

(* suppress messages *)
Off[Attributes::locked];

(* remove ReadProtected attribute *)
If[MemberQ[Attributes[#], ReadProtected],
   ClearAttributes[#, ReadProtected]]& /@
Apply[Unevaluated, ToHeldExpression /@ systemCommands, {1}];

(* all messages *)
allMessages = (Messages @@ #)& /@ (ToHeldExpression[#]& /@
                               DeleteCases[systemCommands, "I"]);
```

```
allMessagesUnevaluated = Unevaluated @@ #& /@ (First /@ Flatten[allMessages
```

```
(* shut off all messages *)
Off /@ allMessagesUnevaluated;
```

```
(* remove inappropriate functions *)
goodSystemCommands = Select[Complement[systemCommands,
  {"Break", "Continue", "ConsoleMessage", "Edit", "FixedPoint",
   "FixedPointList", "$Inspector", "OpenTemporary", "$PrintHoldPattern",
   "Streams", "Remove", "$Epilog", "Return", "Set", "SubValues",
   "Run", "Print", "SetDelayed", "Throw", "$PrintLiteral",
   "ConvertToPostScript", "ArrayRules", "Signature"}],
                                    # === ToString[ToExpression[#]]&];
```

Here is the code for the actual search. We use `StringMatchQ` to recognize the `()`.

```
li = Select[goodSystemCommands, (StringMatchQ[ToString[ToExpression[
                    # <> "[x, " <> "PrecedenceForm[y, 1]]"]], "*(*)*"])&]
```

Now, increasing the second argument of `PrecedenceForm` stepwise and observing when the `()` disappear, we get the corresponding `PrecedenceLevel` (the use of `ReplaceAll` is needed because of the `Hold`-like attribute of many commands).

```
{#, Module[{i = 1}, While[StringMatchQ[ToString[
                            ToExpression[# <> "[x,
                            " <> "PrecedenceForm[y, k]]"] /. {k -> i}],
                                    "*(*)*"], i = i + 1]; i - 1]}& /@ li;
```

To conclude, we now reorder these somewhat.

```
Sort[%, #1[[2]] < #2[[2]]&] // TableForm
```

The second element of the result of `PrintForm[`*expr*`]` contains also the explicit precedence level.

```
{PrintForm[Unevaluated[a @ b]][[2]],
 PrintForm[Unevaluated[a /@ b]][[2]],
 PrintForm[Unevaluated[a @@ b]][[2]],
 PrintForm[Unevaluated[a // b]][[2]],
 PrintForm[Unevaluated[a //@ b]][[2]]}
```

With a knowledge of `PrecedenceLevel`, we now know when to use brackets `()` and can understand the meaning of the written out expression.

```
_!_ == __||__ == __ == __ == __||__ == _!_

FullForm[Hold[_!_ == __||__ == __ == __ == __||__ == _!_]]
```

These are the names of all named characters.

```
allNamedCharacters =
DeleteCases[Select[FromCharacterCode /@ Range[10^5],
             Characters[ToString[FullForm[#]]][[-2]] === "]"&], "]"];
```

Not all of them are operators, many are letter-like forms.

```
Take[allNamedCharacters, -12] // InputForm
```

We extract the operator name corresponding to the character names.

```
characterNames = {#, StringDrop[StringDrop[
       ToString[FullForm[#]], -2], 3]}& /@ allNamedCharacters;
```

Now, we construct *characterFunction*`[x, PrecedenceForm[y, 1]]` to find the names that represent operators.

```
li2 =
Select[characterNames,
         (StringMatchQ[ToString[ToExpression[
           #[[2]] <> "[x, " <> "PrecedenceForm[y, 1]]"]], "*(*)*"])&];
```

Continuing in the same way as above, we now investigate *characterFunction*`[x, PrecedenceForm[y, ` *k*`]]` to determine the precedence.

```
{#, Module[{i = 1},
          While[StringMatchQ[ToString[
                    ToExpression[#[[2]] <> "[x,
                    " <> "PrecedenceForm[y, k]]"] /. {k -> i}],
                  "*(*)*"], i = i + 1]; i - 1]}& /@ li2;
```

Here are the operators together with their precedences.

```
With[{L = Sort[Union[%], #1[[2]] < #2[[2]]&]},
TableForm[Flatten /@ Partition[If[EvenQ[Length[L]], L,
                          Append[L, {" ", " ", " "}]], 2],
          TableSpacing -> {0.5, 1}]]
```

We now turn the messages back on.

```
On /@ allMessagesUnevaluated;
```

```
Off[General::newsym]
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

21. One-Liners

a) The programming objective is to look for every given set of summands that can be fit into the difference between `sum` and the already accumulated number. We do not count the trivial result when all factors are 0.

Here are three possibilities.
The first version uses the construction of an iterator. The multiple iterator is built by using `Unique` to generate the iterator. Variables are constructed as lists, and then `Sequence` removes the outermost curly brackets.
Note that `Evaluate` is necessary in all arguments (body and iterator) of `Table` (because of the attribute `HoldAll`).

```
AllPossibleFactors1[sum_?(TrueQ[# > 0]&),
                    summands_?(VectorQ[#, TrueQ[# > 0]&]&)] :=
Rest[Function[l, Flatten[
 Table[Evaluate[#], Evaluate[Sequence @@
          MapThread[List, {#, Array[0&, {l}],
                  (sum - Drop[FoldList[Plus, 0, MapThread[Times,
                          {#, summands}]], -1])/summands}]]],
                                  l - 1]&[Table[Unique[i], {l}]]][
                                       Length[summands]]]
```

Here is a simple example.

```
AllPossibleFactors1[8, {4, 2, 1}]
```

All resulting sums are less than or equal to 8.

```
{4, 2, 1}.#& /@ %
```

When all summands are bigger than the sum, we get an empty list as the result.

```
AllPossibleFactors1[8, {44, 24, 11}]
```

Now, the question concerning one dollar is calculated.

```
AllPossibleFactors1[100, {25, 10, 5, 1}] // Length
```

The second arrangement uses `Fold` to generate the nesting. Every already-existing sequence of factors is used to determine the iterator for `Range` in the next step.

```
AllPossibleFactors2[sum_?(TrueQ[# > 0]&),
                    summands_?(VectorQ[#, TrueQ[# > 0]&]&)] :=
Rest[Fold[Function[{was, is},
   Flatten[Function[old, Flatten[{old, #}]& /@
    Range[0, (sum - Drop[is, -1].old)/Last[is]]] /@ was, 1]],
Array[{#}&, Floor[sum/First[summands]] + 1, 0],
 Drop[Flatten /@ FoldList[List, {}, summands], 2]]]
```

For comparison, we again calculate the division of the dollar.

```
AllPossibleFactors2[100, {25, 10, 5, 1}] // Length
```

The last version here is a slightly rewritten form of the previous example, which uses `Array` rather than `Range`. Note that for `Array`, the second argument has to be an integer, and so `Floor` (see Chapter 1 of the Numerics volume [302✱]) is necessary here.

```
AllPossibleFactors3[sum_?(TrueQ[# > 0]&),
                    summands_?(VectorQ[#, TrueQ[# > 0]&]&)] :=
Rest[Fold[Function[{was, is},
   Flatten[Function[old, Array[Flatten[{old, #}]&,
      Floor[(sum - Drop[is, -1].old)/Last[is]] + 1, 0]] /@ was, 1]],
Array[{#}&, Floor[sum/First[summands]] + 1, 0],
 Drop[Flatten /@ FoldList[List, {}, summands], 2]]]
```

For a third and last time, the dollar splitting is calculated.

```
AllPossibleFactors3[100, {25, 10, 5, 1}] // Length
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

b) Here is a direct translation of the implementation from Exercise 9.d) in Chapter 5.

```
FerrerConjugate1[l_List] :=
      Drop[Length /@ FixedPointList[DeleteCases[# - 1, 0]&, l], -2]
```

We test, using the two examples from the last chapter.

```
FerrerConjugate1[{6, 3, 2}]

FerrerConjugate1[{2, 2, 2, 2, 2, 1}]
```

Another possibility would be to count the numbers in the list that are greater than 1, 2,…, n_1.

```
FerrerConjugate2[l_List] :=
     Function[i, Count[l, _?(# >= i&)]] /@ Range[First[l]]

FerrerConjugate2[{6, 3, 2}]

FerrerConjugate2[{2, 2, 2, 2, 2, 1}]
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

c) We will call our model of `AppendTo` lowercase `appendTo`. `appendTo` must have the `HoldFirst` attribute.

```
SetAttributes[appendTo, HoldFirst];
```

The model of `AppendTo` evaluates the list in the right-hand side of `Set`, appends the new element, and assigns the result to the name of the list.

```
appendTo[l_, new_] := Set[l, Append[l, new]]
```

Here is a quick check for `appendTo`s behavior.

```
Λ[1] = {1, 2, 3};
```

```
appendTo[Λ[1], 4]
```

```
Λ[1]
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

d) Let us start by implementing the calculation of the products $p_{i_1} \ldots p_{i_j}$. `products` forms all possible products with k factors.

```
products[ps_, k_] := Flatten[
Table[Times @@ ps[[#]],
      Evaluate[Sequence @@ Table[{i[j], If[j == 1, 1, i[j - 1] + 1],
                 Length[ps]}, {j, k}]]]&[Table[i[j], {j, k}]]]
```

Here are all products of five symbols with no powers.

```
Table[products[{a, b, c, d, e}, k], {k, 6}]
```

Using `products`, it is straightforward to implement Meissel's formula.

```
pi[1] = 0;
pi[n_] := With[{ps = Prime[Range[pi[Floor[Sqrt[n]]]]]},
               n - 1 + pi[IntegerPart[Sqrt[n]]] +
                Sum[(-1)^k Plus @@ IntegerPart[n/products[ps, k]],
                    {k, Length[ps]}]]
```

The exercise asked for an implementation with only built-in symbols. This means we must eliminate the `ps`, `pi`, and the iterator variables. For brevity, we will use one-letter built-in symbols. There are seven to choose from.

```
Select[Names["*"], (StringLength[#] === 1 && UpperCaseQ[#])&]
```

For the function `pi`, we will use `PrimePi`. `PrimePi` is the built-in function that calculates the number of primes less than or equal to its argument. To not interfere with its built-in meaning, we give it an option. We use a string as the option value. So, we end with the following implementation.

```
Unprotect[PrimePi];
PrimePi[1, Method -> "Meissel"] = 0;

PrimePi[N_, Method -> "Meissel"] :=
Module[{C, D, K}, Function[O,
N - 1 + PrimePi[IntegerPart[Sqrt[N]], Method -> "Meissel"] +
          Sum[(-1)^K Plus @@ IntegerPart[N/Flatten[Table[Times @@ O[[#]],
      Evaluate[Sequence @@ Table[{C[D], If[D == 1, 1, C[D - 1] + 1],
                 Length[O]}, {D, K}]]]&[Table[C[D], {D, K}]]]],
               {K, Length[O]}]][Prime[Range[
               PrimePi[Floor[Sqrt[N]], Method -> "Meissel"]]]]]
```

Here is a quick check that only built-in symbols were used.

```
Union[Context /@ Cases[DownValues[PrimePi], _Symbol, {-1}, Heads -> True]]
```

The values calculated by our `PrimePi` agree with the values of the built-in version.

```
PrimePi[1000, Method -> "Meissel"]
```

```
PrimePi[1000]
```

The above implementation could be slightly improved for efficiency. Instead of multiplying all numbers for each product, we could carry out this recursively.

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

e) The implementation of the p_n is straightforward. We first generate a list of the x_k and the permutations σ. Then we map the function μ_k to the permutations, then multiply and sum the result. Finally, we factor the result.

```
permutationPolynomial[n_Integer, x_] :=
Function[xs, Factor[Plus @@ (Function[p, Times @@
(xs^Array[Function[j, Count[Drop[p, j], _?(# < p[[j]]&)]], n])] /@
                                    Permutations[Range[n]])]][Array[x, n]]
```

Here are the polynomials p_1 to p_8 explicitly calculated.

```
permutationPolynomial[1, x]

permutationPolynomial[2, x]

permutationPolynomial[3, x]

permutationPolynomial[4, x]

permutationPolynomial[5, x]

permutationPolynomial[6, x]

permutationPolynomial[7, x]

permutationPolynomial[8, x]
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

f) Here is a straightforward implementation of this differential identity.

```
diffId[k_Integer, p_Integer] := (Sum[
  (* make body of sum *) Evaluate[
  Product[D[f[x]^n[j]/n[j]!, {x, n[j] - 1}], {j, p}]*
  D[f[x]^(k - Sum[n[j], {j, p}])/(k - Sum[n[j], {j, p}])!,
     {x, k - Sum[n[j], {j, p}] - 1}]],
  (* make iterators *)
  Evaluate[Sequence @@ Transpose[{Table[n[j], {j, p}],
           FoldList[Subtract, k - 1, Table[n[j], {j, p - 1}]]}]]] -
(p + 1) (k - 1)!/(k - 1 - p)! D[f[x]^k/k!, {x, k - p - 1}] // Expand)  /;
                                                                0 < p < k
```

Next, we check all allowed p and k for $k \leq 12$.

```
Table[diffId[k, p], {k, 12}, {p, k - 1}]
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

g) Here is again a straightforward implementation of the identity. We first calculate the sequence of traces (keeping only the highest power of the matrix). Then, we form the new matrix and calculate its determinant.

```
det[𝔸_?MatrixQ] := Function[n, 1/n! Det[
Function[𝕒, Array[Which[#1 >= #2, 𝕒[[#1 - #2 + 1]], #2 == #1 + 1, #1,
                  True, 0]&, {n, n}]][Last /@ (* traces of powers *)
         FoldList[{#, Tr[#]}&[#2.#1[[1]]]&, {𝔸, Tr[𝔸]},
                              Table[𝔸, {n - 1}]]]]]][Length[𝔸]]
```

For a "random" matrix, we again check that the results of inverse agree with the left-hand side, meaning `Det`. Of course, `det` is much slower.

```
𝔸 = With[{n = 12}, Table[(i + j)/(i j + 1), {i, n}, {j, n}]];

{(det𝔸1 = Det[𝔸]); // Timing, (det𝔸2 = det[𝔸]); // Timing,
 det𝔸1 - det𝔸2}
```

For a similar expression for the discriminant of the characteristic polynomial of a matrix, see [235★].

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

h) Here is a straightforward implementation of the identity. We calculate the characteristic polynomial only once.

```
inverse[𝔸_?MatrixQ] := (-1/#1 (#2 /. ξ^k_. :> MatrixPower[𝔸, k - 1]))& @@
 (Function[cp, {#, cp - #}&[cp /. ξ -> 0]][CharacteristicPolynomial[𝔸, ξ]])
```

For a "random" matrix, we check that the results of `inverse` agree with the results of the built-in function `Inverse`. Of course, `inverse` is much slower.

```
𝔸 = With[{n = 12}, Table[(i + j)/(i j + 1), {i, n}, {j, n}]];

{(inv𝔸1 = Inverse[𝔸]); // Timing, (inv𝔸2 = inverse[𝔸]); // Timing,
 inv𝔸1 - inv𝔸2 // Flatten // Union}

Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

i) It is straightforward to implement this product expansion. The optional argument s is a potential simplifier.

```
productForm[f_, {z_, z0_, o_}, s_:Identity] :=
Module[{ξ}, (Times @@
MapIndexed[#1^(Log[z/ξ]^(#2[[1]] - 1)/(#2[[1]] - 1)!)&,
           NestList[s[Exp[ξ D[Log[#], ξ]]]&, f[ξ], o]] /. ξ -> z0)]
```

Here are the first factors for a general f and a general expansion point.

```
productForm[f, {z, z, 3}]
```

$\Pi_{12}(\cos(\pi/2), 1)$ is a relatively large expression. But it approximates the true result relatively purely.

```
cosProduct12 = productForm[Cos, {Pi/2, 1, 12}];
{ByteCount[cosProduct12]/10.^6 MB, N @ cosProduct12}

Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

j) Here is a one-liner forming all binary function-based expressions. Recursively we simply form all pairs of adjacent neighbors.

```
allBinaryCompositions[argList_, f_] :=
Nest[Union[Flatten[Table[
          Join[Take[#, k - 1], {f[#[[k]], #[[k + 1]]]},
               Take[#, {k + 2, Length[#]}]],
                          {k, Length[#] - 1}]& /@ #, 1]]&,
    {argList}, Length[argList] - 1] // Flatten
```

Here are two examples. We use four and five arguments.

```
allBinaryCompositions[{a, b, c, d}, f]

allBinaryCompositions[{a, b, c, d, e}, f]
```

The number of different expressions obtained using `allBinaryCompositions` are the Catalan numbers [285✱].

```
Table[Length @ allBinaryCompositions[Range[n], f], {n, 2, 12}]

Needs["DiscreteMath`CombinatorialFunctions`"]
Table[CatalanNumber[n - 1], {n, 2, 12}]
```

To count how frequently we have k consecutive closing ')', we transform the expressions into a string and then count consecutive closing '`]`'.

```
bracketCounter[expr_, allBracketStrings_] :=
MapIndexed[#1/#2[[1]]&, Reverse[Length[First[#]]& /@
(* start with longest sequence and count backwards *)
FoldList[(* which are new? *)
        {Complement[#2, #1[[2]]], Union[Join[#1[[2]], #2]]}&,
        {(* new *){}, (* occurred already *) {}},
        Flatten /@ Reverse[(* positions of k consecutive ] *)
        (Range @@@ StringPosition[ToString[expr], #])& /@
         allBracketStrings]]]]
```

Here is an example.

```
bracketCounter[f[f[a, f[f[b, c], d]], e], {"]", "]]", "]]]"}]
```

For 10 symbols, we obtain the following distribution for the closing brackets.

```
allBracketStrings = Table[StringJoin[Table["]", {k}]], {k, 10}];
Plus @@ (bracketCounter[#, allBracketStrings]& /@
                      allBinaryCompositions[Range[10], f])
```

Now, we form all possible powers of i. To identify numerically equal powers, we numericalize to high precision and then reduce the number of digits to allow `Sort` to identify equal real parts. This yields 15 different numerical values out of the 37 starting expressions.

```
identicalPowers = {N[#[[1, 1]]], Last /@ #}& /@
Split[Sort[{N[(* high-precision numericalization *) N[#, 1000],
              (* form lower precision number *) 100], #}& /@
      allBinaryCompositions[Table[I, {k, 6}], Power]],
     First[#1] == First[#2]&];
```

Here are the numerically identical, but structurally different power towers.

```
Map[(# /. List -> Equal)&,
    HoldForm /@ Select[Last /@ identicalPowers, Length[#] > 1&]] //
                                        TableForm // TraditionalForm
```

Using `Simplify`, or even the stronger function `FullSimplify` (to be discussed in the Symbolics volume [303✱]), does not allow to show the correctness of all of the above equalities. The function `ComplexExpand` (also to be discussed in the Symbolics volume [303✱]) can show the correctness of the found identities.

```
{Simplify[#], FullSimplify[#], ComplexExpand[#]}&[
                 Equal @@ identicalPowers[[6, 2]]]
```

We end with a visualization. For 13 arguments that are powers of i and f=`Power`, as well for a random complex number and f = arctan, we show all resulting expressions in the complex plane.

```
(* form compositions for symbolic z and f *)
abcList13 = allBinaryCompositions[Table[z^k, {k, 13}], f];
Length[abcList13]

(* form compositions for concrete z and f *)
Internal`DeactivateMessages[
 abcLists13N = (DeleteCases[#, $Aborted]& @
   ((* skip calculations that produce too large intermediate numbers *)
    Function[abc, TimeConstrained[abc /. {z -> #1, f -> #2}, 1]] /@
                                       Take[abcList13, All]))& @@@
              {{1. I, Power}, {-0.0986423 - 0.0046093 I, ArcTan}}];
```

```
(* no messages from too large numbers *) Off[Graphics::gptn];

(* show the two sets of numbers in the complex plane *)
Show[GraphicsArray[Internal`DeactivateMessages @
 Graphics[{PointSize[0.001], Point[N[{Re[#], Im[#]}]]& /@ #1},
         Frame -> True, PlotRange -> #2]& @@@
         Transpose[{abcLists13N, {{{-3, 3}, {-3, 3}},
                                  {{-3, 1}, {-1, 1}/20}}}]]]
```

For a *Mathematica* implementation of the four fours problems, see [172✱].

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

k) To motivate the implementation of `KolakoskiSequence` below, we start with a straightforward procedural way to calculate n terms of the Kolakoski sequence. `KolakoskiP` start by preparing a list `l` of two leading twos and $n-1$ zeros to be filled in. We then step through the list `l` and add elements at the end according to earlier elements that indicate the run length. The construction `k = 3 - k` switches between ones and twos.

```
KolakoskiP[n_] :=
Module[{l, c, k, p, t},
 (* list to be filled in *)
 l = Table[0, {n + 1}];
 l[[1]] = 2; l[[2]] = 2;
 c = 3; (* inserting position *)
 k = 2; (* element of l *)
 p = 2; (* extracting position *)
 (* now add elements *)
 While[c <= n,
       t = l[[p++]];
       k = 3 - k;
       If[t === 1, l[[c++]] = k, l[[c++]] = k; l[[c++]] = k]];
 (* return first n elements *)
 Take[l, n]]
```

The function `runLengthPropertyQ` checks if the list *l* is a Kolakoski sequence.

```
runLengthPropertyQ[l_] :=
With[{ℓ = Length /@ Split[l]}, Take[l, Length[ℓ]] === ℓ]
```

Here are the first 20 numbers of the Kolakoski sequence.

```
KolakoskiP[20]
```

The last sequence, as well as its continuation as returned by `runLengthPropertyQ` is the Kolakoski sequence.

```
{runLengthPropertyQ[%], runLengthPropertyQ[KolakoskiP[10^5]]}
```

Rewriting now the above function `KolakoskiP` in a functional way leads to the following one-liner `KolakoskiSequence`. The `While` is replaced by a `NestWhile`, and the `Table` by `Array` to avoid any named variables. The equivalent to the part assignments to `l` is now the `ReplacePart` construction. And then recursively updated variables `c`, `k`, and `p` are parts of a list that are updated in each `NestWhile` step.

```
KolakoskiSequence[n_Integer?Positive] :=
NestWhile[(If[#1[[#4]] === 1,
              {ReplacePart[#1, 3 - #3, #2],
               #2 + 1, 3 - #3, #4 + 1},
              {ReplacePart[#1, 3 - #3, {{#2}, {#2 + 1}}],
               #2 + 2, 3 - #3, #4 + 1}]& @@ #)&,
          {ReplacePart[Array[0&, n + 1], 2, {{1}, {2}}], 3, 2, 2},
          (#[[2]] <= n)&][[1]] // Take[#, n]&
```

The first 1000 elements of the Kolakoski sequence are calculated by `KolakoskiSequence` within a fraction of a second.

```
(l = KolakoskiSequence[10^3]); // Timing
```

```
(* correctness check *) runLengthPropertyQ[l]
```

To calculate many elements of the Kolakoski sequence quickly (more than a million per second on a fast computer), one would use a compiled version of the above procedural code. We will discuss the function `Compile` in Chapter 1 of the Numerics volume [302✱].

```
KolakoskiSequenceCompiled =  Compile[{{n, _Integer}},
Module[{l = Table[0, {n + 1}], c = 3, k = 2, p = 2, t},
       l[[1]] = 2; l[[2]] = 2;
       While[c <= n, t = l[[p++]];  k = 3 - k;
            If[t === 1, l[[c++]] = k, l[[c++]] = k; l[[c++]] = k]];
       Take[l, n]]];
```

Interestingly, for the Kolakoski sequence (prefaced with 1) there exists a real number $\gamma = 0.3496655\ldots$, such that a normal continued fraction formed from the sequence agrees with the number whose base γ digits are the sequence itself [294✱].

$$\cfrac{1}{1+\cfrac{1}{2+\cfrac{1}{2+\cfrac{1}{1+\cfrac{1}{1+\cfrac{1}{2+\cdots}}}}}} = 1\,\gamma + 2\,\gamma^2 + 2\,\gamma^3 + 1\,\gamma^4 + 1\,\gamma^5 + 2\,\gamma^6 + \cdots$$

The following two inputs confirm this amazing identity.

```
KolakoskiCF = N[#, 50]& @
   FromContinuedFraction[KS = Join[{0, 1}, KolakoskiSequence[100]]]
```

```
With[{γ = 0.34966558689091838185601052042540566151100382812527 6},
     KS.(N[γ, 200]^Range[0, Length[KS] - 1])]
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

l) A straightforward implementation would be along the following lines. Calculating the $\sigma_k(t)$ is straightforward. Replacing the $x(t)$, $y(t)$, and $z(t)$ by one yields automatically the sum of all coefficients.

```
coefficientSum[n_] :=
Module[{x, y, z, τ, σ},
       {x'[t], y'[t], z'[t]} = {y[t] z[t], x[t] z[t], x[t] y[t]};
       σ[0] = x[t];
       σ[k_] := σ[k] = D[σ[k - 1], t];
       σ[n] /. _[t] -> 1]
```

Here is a quick check for $n = 10$.

```
{coefficientSum[10], 10!}
```

Now, we rewrite the above function to avoid the use of any built-in symbol. We replace the explicit definitions for the three derivatives with replacement rules and we carry out the recursive calculation of the $\sigma_k(t)$ using `NestList`. And instead of the user symbols x, y, z, and t, we use just built-in functions that do not have any nontrivial evaluation rules for any number and kind of arguments. Such functions are, for instance, attributes. Here we use `HoldFirst`, `HoldRest`, `HoldAll`, and the symbol `D` for *t* above. Here is the resulting function `factorialSumTest`.

```
factorialSumTest = ((NestList[Expand[D[#, D] /.
      {HoldFirst'[D] -> HoldRest[D] HoldAll[D],
       HoldRest'[D] -> HoldFirst[D] HoldAll[D],
       HoldAll'[D] -> HoldFirst[D] HoldRest[D]}]&,
       HoldFirst[D], #] /. _[D] -> 1) == Range[0, #]!)&;
```

Because of the use of `NestList`, we can now check all *n* less than 100 at once in just a few seconds.

```
factorialSumTest[100] // Timing
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

m) Here is a straightforward implementation. After generating all permutations using `Permutations`, we split each permutation into pairs of adjacent elements. We then just count the number of pairs of the form $\{j_i, j_i + 1\}$ and return a list with elements of the form {*numberOfIncreasing2Sequences*, *numberOfPermutations*}.

```
countIncreasingTwoSequence[n_Integer?Positive] :=
{First[#], Length[#]}& /@
     Split[Sort[(Count[(Subtract[##] == -1)& @@@
           Partition[#, 2, 1], True])& /@ Permutations[Range[n]]]]
```

Here is an example.

```
countIncreasingTwoSequence[8]
```

The following input calculates the number of increasing two-sequences using a closed-form formula [151✱].

```
Module[{n = 8, 𝒟},
       𝒟[k_] := k! Sum[(-1)^j/j!, {j, 0, k}] (* Gamma[k + 1, -1]/E *);
       Table[{k, Binomial[n, k] 𝒟[n - k + 1]/n}, {k, 0, n}]]
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

22. Precedences

a) In the first example, not much interesting happens. The pure function `Function[x, Hold[x], {Listable}]` is applied to the argument `Hold[{1 + 1, 2 + 2, 3 + 3}]`. Because the argument has the head `Hold`, the `Listable` attribute of the pure function cannot do anything and the result is just the argument enclosed in an additional `Hold`.

```
Function[x, Hold[x], {Listable}] @ Hold[{1 + 1, 2 + 2, 3 + 3}]
```

In the second example, the pure function `Function[x, Hold[x], {Listable}]` is applied (this time in the sense of `Apply`) to `Hold[{1 + 1, 2 + 2, 3 + 3}]`. So the head `Hold` gets replaced by `Function[x, Hold[x], {Listable}]`. Now, the argument `{1 + 1, 2 + 2, 3 + 3}` evaluates first to `{2, 4, 6}` and then the pure function with the `Listable` attribute comes to work and applies `Hold` to every element of this list.

```
Function[x, Hold[x], {Listable}] @@ Hold[{1 + 1, 2 + 2, 3 + 3}]
```

In the third example, the pure function `Function[x, Hold[x], {Listable, HoldAll}]` is applied to `Hold[{1 + 1, 2 + 2, 3 + 3}]`. The additional attribute `HoldAll` of the pure function does not matter here because the argument is already wrapped in `Hold` and the result is the same, as in the first example.

```
Function[x, Hold[x], {Listable, HoldAll}] @ Hold[{1 + 1, 2 + 2, 3 + 3}]
```

In the fourth example, the function `Function[x, Hold[x], {Listable, HoldAll}]` is applied (this time again in the sense of `Apply`) to `Hold[{1 + 1, 2 + 2, 3 + 3}]`. But now the pure function has the attribute

HoldAll, so its argument stays unevaluated and the Listable attribute can come to work to give {Hold[1 + 1], Hold[2 + 2], Hold[3 + 3]}.

```
Function[x, Hold[x], {Listable, HoldAll}] @@ Hold[{1 + 1, 2 + 2, 3 + 3}]
```

In the fifth example, the argument of the pure function Function[x, Hold[x], {Listable, HoldAll}] is a more complicated expression. Because of the HoldAll attribute, nothing happens again and the result is just the whole argument wrapped in an outer Hold.

```
Function[x, Hold[x], {Listable, HoldAll}] @
                     (#& @@ Hold[{1 + 1, 2 + 2, 3 + 3}])
```

In the sixth example, the pure function Function[x, Hold[x], {Listable}] is applied (here again in the sense of Apply) to its argument. The argument evaluates to {2, 4, 6}.

```
#& @@ Hold[{1 + 1, 2 + 2, 3 + 3}]
```

Now applying the pure function Function[x, Hold[x], {Listable}] results in Function[x, Hold[x], {Listable}][2, 4, 6]. Because the pure function takes only one argument, the first argument gets taken out and the result is Hold[2].

```
Function[x, Hold[x], {Listable}] @@ (#& @@ Hold[{1 + 1, 2 + 2, 3 + 3}])
```

The seventh example is similar to the fifth one, but this time there are no explicit parentheses for grouping. Because @ binds here more strongly than @@ (binding of @ and @@ works from right to left), the structure of the expression is now the following.

```
FullForm[Hold[Function[x, a] @ #& @@ y]]
```

The result of evaluating the first argument of Apply is the pure function Function[x, Hold[x], {Listable, HoldAll}][#1]&. This then gets applied (in the sense of Apply to Hold[1 + 1, 2 + 2, 3 + 3]. The outer pure function has no HoldAll attribute, so the resulting expression is Function[x, Hold[x], {Listable, HoldAll}][{2, 4, 6}], which finally gives {Hold[2], Hold[4], Hold[6]}.

```
Function[x, Hold[x], {Listable, HoldAll}] @
                      #& @@ Hold[{1 + 1, 2 + 2, 3 + 3}]
```

The eighth example has the following structure.

```
FullForm[Hold[Function[x, x] @ Function[y, y] @@ a]]
```

First, the two arguments of Apply get evaluated. The first argument results in substituting the whole pure function Function[x, Hold[x], {Listable, HoldAll}] as the x in the Hold of the outer one. The result is the following expression.

```
Function[x, Hold[x], {Listable, HoldAll}] @
    Function[x, Hold[x], {Listable, HoldAll}]
```

The second argument of Apply is just Hold[1 + 1, 2 + 2, 3 + 3], which, because of the Hold, stays unchanged. Then, Apply comes to work and replaces the Hold of the second argument by the first argument. Because of the Hold wrapped around the head of this expression, the evaluation ends here.

```
Function[x, Hold[x], {Listable, HoldAll}] @
    Function[x, Hold[x], {Listable, HoldAll}] @@
                         Hold[{1 + 1, 2 + 2, 3 + 3}]
```

The ninth example contains the @ and @@ interchanged in comparison with the last example. Now, the expression to be analyzed has the following structure.

```
FullForm[Hold[Function[x, x] @@ Function[y, y] @ a]]
```

This time the stronger binding `@` (it is leftmost) has the result that the second argument of `Apply` is now `Function[x, Hold[x], {Listable, HoldAll}] @ Hold[{1 + 1, 2 + 2, 3 + 3}]`. The result of evaluating this is `Hold[Hold[{1 + 1, 2 + 2, 3 + 3}]]`. Now, the first argument of `Apply`, the pure function `Function[x, Hold[x], {Listable, HoldAll}]`, replaces the head `Hold` of `Hold[Hold[{1 + 1, 2 + 2, 3 + 3}]]`, to give `Function[x, Hold[x], {Listable, HoldAll}][Hold[{1 + 1, 2 + 2, 3 + 3}]]`. This expression finally evaluates again to `Hold[Hold[{1 + 1, 2 + 2, 3 + 3}]]`.

```
Function[x, Hold[x], {Listable, HoldAll}] @@
    Function[x, Hold[x], {Listable, HoldAll}] @ Hold[{1 + 1, 2 + 2, 3 + 3}]
```

The tenth and last example has the following structure.

```
FullForm[Hold[Function[x, x] @@ Function[y, y] @@ a]]
```

This time the second argument of the outer `Apply` has itself the head `Apply`. This second argument evaluates to `{Hold[1 + 1], Hold[2 + 2], Hold[3 + 3]}`. Now, the outer `Apply` comes to work and gives `Function[x, Hold[x], {Listable, HoldAll}][Hold[1 + 1], Hold[2 + 2], Hold[3 + 3]]`, which finally evaluates to `Hold[Hold[1 + 1]]`.

```
Function[x, Hold[x], {Listable, HoldAll}] @@
    Function[x, Hold[x], {Listable, HoldAll}] @@ Hold[{1 + 1, 2 + 2, 3 + 3}
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

b) Obviously, what must be avoided is that the `Print[localVar];` is carried out without changing `localVar` to 11. This can be achieved using postfix notation with a construction of the form `Print[localVar] // Hold`. After `Print[localVar]` has been wrapped in `Hold`, we must change the value of `localVar` and then carry out the `Print` statement. Here are ways to do this.

```
localVar = 11;
Block[{localVar = 1},
      Print[localVar]; //
         Hold //
         (MapAt[Function[p, localVar = 11; p, {HoldAll}], #, {1}]&) //
         ReleaseHold]
```

```
localVar = 11;
Block[{localVar = 1},
      Print[localVar]; // Hold // (localVar = 11; #&) // ReleaseHold]
```

We could also explicitly replace the `1` by the needed `11`.

```
localVar = 11;
Block[{localVar = 1},
      Print[localVar]; // Hold //
      (# /. HoldPattern[localVar] -> 11 &) // ReleaseHold]
```

The last construction also works for `With`.

```
localVar = 11;
With[{localVar = 1},
     Print[localVar]; // Hold // (# /. 1 -> 11&) // ReleaseHold]
```

We could also use a more dirty way (this means taking into account issued messages) to achieve the 11 printed. Both `Block` and `Module` expect two arguments. If we call them with more than two arguments, no built-in code causes any nontrivial evaluation. So, we would just apply `Evaluate` in postfix notation to the `Print` statement.

```
Block[{localVar = 1}, Print[localVar]; // Evaluate, thirdArgument]
```

The message can be avoided by using `Off` in the evaluated third argument of `Block`.

```
Block[{localVar = 1}, Print[localVar]; // Evaluate,
      Evaluate[Off[Block::"argrx"]]]

With[{localVar = 1}, Print[localVar]; // Evaluate, thirdArgument]
```

Σ (* session summary *) **TMGBs`PrintSessionSummary[]**

23. Puzzles

a) Using `FullForm`, we can see the grouping better. (Be aware of the difference the spacing before the 10 and the 11 makes.)

```
FullForm[Hold[1 @ 2 @@ 3 / 4 /@ 6 //@ 7 || 8 | 9 /. 10 /.11]]
```

The following subexpressions give a nontrivial evaluation.

```
Apply[1[2], 3]

MapAll[6, 7]

Times[10, Power[0.11, -1]]
```

So, we finally have the following result.

```
1 @ 2 @@ 3 / 4 /@ 6 //@ 7 || 8 | 9 /. 10 /.11

FullForm[%]
```

Σ (* session summary *) **TMGBs`PrintSessionSummary[]**

b) For "ordinary" input, `Function[`*bodyWithSlot*`]` and `Function[`*x*`, ` *bodyWithx*`]` will behave in the same way. So for *factor*=*α*, we get the same output from the following functions.

```
scaledReversedShiftedListV1[factor_, list_List] :=
    Function[Join[factor #, Reverse[factor/2 #]]][list]

scaledReversedShiftedListV2[factor_, list_List] :=
    Function[x, Join[factor x, Reverse[factor/2 x]]][list]

scaledReversedShiftedListV1[α, {1, 2, 3}]

scaledReversedShiftedListV2[α, {1, 2, 3}]
```

If we use `x` as the first argument, we still get the same result. The `x` in the first argument of `Function` is properly renamed.

```
scaledReversedShiftedListV1[x, {1, 2, 3}]

scaledReversedShiftedListV2[x, {1, 2, 3}]
```

The dummy variable `x` inside `Function` was replaced by `x$` so that there is no naming collision with the other `x`. Holding the right-hand side of the definitions above shows this nicely.

```
showScreening[factor_, list_List] :=
    Hold[Function[x, Join[factor x, Reverse[factor/2 x]]][list]]

showScreening[x, {1, 2, 3}]
```

Because `Slot` variables cannot be locally renamed, the two functions give different results for *factor* = `#`.

```
scaledReversedShiftedListV1[#, {1, 2, 3}]

scaledReversedShiftedListV2[#, {1, 2, 3}]
```

Σ (* session summary *) **TMGBs`PrintSessionSummary[]**

c) The problem is to predict what will be the *Mathematica* meaning of `1.....` (*n* periods).

`1.` is the real number 1.

`1..` cannot be parsed

`1...` means `Repeated[1.]`

`1....` means `RepeatedNull[1.]`.

More points, then, repeat the above listing by forming nested structures.

```
{#, InputForm[ToExpression @ #],
    FullForm[ToExpression @ #]}& /@
Table["1" <> Table[".", {i}], {i, 1, 11}] // TableForm
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

d) The `Unevaluated` prevents `Times[2, 2, 2]` from evaluating to 8; instead `Times[2, 2, 2]` is given unevaluated to `Apply`, which changes the head `Times` to the head `Power`, and the result of `Power[2, 2, 2]` is 16.

```
Power @@ Unevaluated[Times[2, 2, 2]]
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

e) The result will be 2 and a message will be issued.

```
Power[Delete @@ Cos[Sin[2], 0]]
```

`Cos[Sin[2], 0]` calls the `Cos` function with two arguments. No built-in rules exist for this case; a message is issued and the expression returns unchanged. Then, `Delete` gets applied to this expression, meaning `Delete[`↵ `Sin[2], 0]` is formed. With the level specification 0, `Delete` will delete the head. This means `Sequence[2]` is the result. Finally, `Power[2]` evaluates to `2`.

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

f) First, the `NestList` part is carried out. The function applied by `NestList` at every step is the following: Take the outer product of the argument with itself and return the resulting nested list. The starting list is the list `{1., 2}`.

The application of `Outer` is carried out three times. After the first application, we have the following nested list.

```
Outer[List, {1., 2}, {1., 2}]
```

After the second application, we have this result.

```
Outer[List, %, %]
```

In every application of `Outer`, the nesting level rises from n to $2n+1$ ($2n$ from forming the outer product and 1 from the newly created lists at level {-2}).

The number of elements `Length[Flatten[#]]` is equal to 2^n, where n is the length of the result of applying `Dimensions` to the expression.

So, we finally have our result.

```
{Dimensions[#], Length[Flatten[#]]}& /@
                  NestList[Outer[List, #, #]&, {1., 2}, 3]
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

g) Here is a held expression and a first attempt to replace the sums.

```
Hold[g[1 + 1, 2 + 2 + 2]] /. p_Plus :> Length[p]
```

Because of the `Hold` around the expression and the `HoldRest` attribute of `RuleDelayed`, the result does not contain evaluated versions of `Length`. The following two approaches also do not succeed. Now, the right-hand side of the rules evaluated before the actual `Plus` expression is substituted.

```
Hold[g[1 + 1, 2 + 2 + 2]] /. p_Plus :> Evaluate[Length[p]]
```

```
Hold[g[1 + 1, 2 + 2 + 2]] /. p_Plus -> Length[p]
```

We can achieve the evaluation we are looking for by using `Condition` inside the right side of the rule and evaluating the unevaluated version of `Plus`.

```
Hold[g[1 + 1, 2 + 2 + 2]] /. HoldPattern[p_Plus] :>
            With[{eval = Length[Unevaluated[p]]}, eval /; True]
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

h) `Infinity` is a symbol. As such, `Block` will scope it and treat it as a local symbol with no built-in meaning. This means `Infinity-Infinity` will be treated like $c-c$ and the result is 0.

```
Block[{Infinity}, Apply[Subtract, {Infinity, Infinity}]]
```

Without the scoping of `Block`, we would obtain `Indeterminate`.

```
Apply[Subtract, {Infinity, Infinity}]
```

Crucial in the behavior above is the fact that `Infinity` did not evaluate to `DirectedInfinity[1]`.

```
Hold[Infinity] // FullForm
```

```
Infinity // FullForm
```

`DirectedInfinity[1]` is not a symbol and cannot be used as a local variable inside `Block`.

```
Block[{DirectedInfinity[1]},
      Apply[Subtract, {DirectedInfinity[1], DirectedInfinity[1]}]]
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

i) We evaluate the first two inputs.

```
inherit[fNew_, fOld_] :=
CompoundExpression[
  SetAttributes[fNew, Attributes[fOld]];
  Options[fNew] = Options[fOld];
  (#[fNew] = (#[fOld] /. fOld -> fNew))& /@
  {NValues, SubValues, DownValues, OwnValues, UpValues, FormatValues}]
```

```
SetAttributes[f, {Listable}];
f[x_Plus] := Length[Unevaluated[x]];
```

Let us study the function `inherit`. It will add all of the definitions (meaning its attributes, options, and various values) given for a symbol `fOld` to the symbol `fNew`. (This means the function `fNew` will inherit the properties of `fOld` [292✱]. For a detailed discussion of inheritance in *Mathematica* see [126✱].)

```
inherit[fNew_, fOld_] :=
CompoundExpression[
 (* take over attributes *)
 SetAttributes[fNew, Attributes[fOld]];
 (* take over options *)
 Options[fNew] = Options[fOld];
 (* take over all definitions *)
 (#[fNew] = (#[fOld] /. fOld -> fNew))& /@
 {NValues, SubValues, DownValues, OwnValues, UpValues, FormatValues}]
```

```
SetAttributes[f, {Listable}];
f[x_Plus] := Length[Unevaluated[x]];
```

Here, we transfer the definitions of `f` to `𝕗`.

```
inherit[𝕗, f];
```

```
??𝕗
```

Now, let us look at the `Module`. The local variable is the symbol `f`. This means at runtime `Module` will create a variable `f$`*number*. The first statement of the body of the `Module` transfers the definitions of `f` to `f$`*number*. The `ToExpression["f"]` creates a symbol `f` different from the local to `Module` variable `f$`*number* and identical to the variable `f` we already gave a definition for. Then, `f$`*number* gets the additional attribute `HoldAll`. Then, a further definition for `f$`*number* for the case of multiple integer arguments is made. Finally, `f$`*number* `@@ f$`*number*`[{1 + 1, 2 + 2}]` gets carried out. According to the `Listable` and `HoldAll` attribute, `f$`*number*`[{1 + 1, 2 + 2}]` is transformed to `{f$`*number*`[1 + 1], f$`*number*`[2 + 2]}`. Then, the inherited definition `f$`*number*`[x_Plus] := Length[Unevaluated[x]]` fires and we get `{2, 2}`. Now, `f$`*number* gets applied yielding `f$`*number*`[2, 2]`. The definition `f$`*number*`[i__Integer] = i^2` fires and we obtain `Sequence[2, 2]^2`. The last expression evaluates to `Power[2, 2, 2]`, which finally evaluates to 16.

```
Module[{f},
       inherit[f, ToExpression["f"]];
       SetAttributes[f, HoldAll];
       f[i__Integer] = i^2;
       f @@ f[{1 + 1, 2 + 2}]]
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

j) Three messages are generated. The first message is issued from `Block` because it is unable to localize a symbol with the attribute `Locked`.

```
Block[{I = 1}, I^2]
```

The second message is generated after the evaluation of `Evaluate[...]` in the first argument of `Block` where an assignment to a symbol with the attribute `Protected` is tried.

```
I = 1
```

The third message is again from `Block`. This time the first argument of `Block` does not have the expected structure. There is no built-in rule for `Block` for this case, and as a result `Block[{1}, -1]` is returned.

```
Block[{1}, -1]
```

For comparison, we evaluate the original input.

```
Evaluate //@ Block[{I = 1}, I^2]
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

k) For all "ordinary" expressions `First[`*expr*`]` and *expr*`[[1]]` will give identical results. They will return different results when *expr*, say, has the head `Sequence`.

```
expr = Sequence[];
{First[expr], expr[[1]]}
```

```
expr = Sequence[1];
{First[expr], expr[[1]]}
```

expr could also contain assignments that behave differently inside `First` and `Part`.

```
expr := (a /: First[a] = 1; a)
{First[expr], expr[[1]]}
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

l) The first input makes a definition for `f`. The right-hand side of the definition is a `Block` construct. The local variable of the `Block` is α. When entering the `Block`, the variable α gets initialized with the value `Not[TrueQ[α]]`. This means that if α has the value `True`, α will become `False`; when α already has the value `False`, α will become `True`; and when α is neither `True` or `False` then α will become `True`. The body of the `Block` then carries out the calculation `f[x + 1]` under the condition α.

```
f[x_] := Block[{α = Not[TrueQ[α]]}, f[x + 1] /; α]
```

The second input starts with the calculation of `f[0]`. Initially α does not have a value, so the α on the left-hand side of the first argument of `Block` evaluates to `True`. As a result, `f[0 + 1] /; α` in the body of `Block` evaluates to `f[1]`. The evaluation now continues (still being inside the originally entered `Block`) with `f[1]`. A new `Block` is opened and the *new* local variable α now gets initialized to `False`, because the α in `Not[TrueQ[α]]` is the one from the first `Block` with the value `True`. As a result, the condition in `f[1 + 1] /; α` evaluates to `False`, and `f[1]` is the result of evaluating `f[0]`. After the argument `f[0]` in `Apply[f, f[0]]` has been evaluated, `Apply` goes into effect and `f[1]` evaluates to `f[1]`. Now again the definition for `f[x]` fires, and repeating the steps from above it evaluates to `f[2]`. This is the result returned.

```
f @@ f[0]
```

Using `Trace`, the described steps are easy to identify.

```
Trace[f @@ f[0]]
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

m) First, let us be clear about the grouping of the body of the two `Module`s that contain a mixture of prefix, postfix, and infix notation.

```
Hold[COrSet @@ f[x_] ~ SetOrC ~ x // f[x]&] // FullForm
```

The last output shows that after evaluating the head `Function[F]` the expression `SetOrC[f[Pattern[x,` `Blank[]]],x]` gets evaluated. Then `COrSet` is applied to the result and finally the body of the pure function `F` is evaluated. Inside the first `Module`, a definition for `f` is created. Although `x` is a variable declared local to `Module`, the presence of the pattern `x_` in `Set` makes the `x` local to `Set`. As a result, we have a definition of the form *f*[*x*_]=*x*. This definition evaluates and then `C` gets applied to its result `x`. Finally `f[C]` gets evaluated using the just set-up definition for `f`. This gives `C`.

```
Module[{x = D, f}, C @@ f[x_] ~ Set ~ x; DownValues[f]]
```

In evaluating the second `Module` things go differently. First `C[f[x_], x]` evaluates to `C[f[x$`*number*`_], D]`. But because this time `x_` does not appear in a scoping construct the right-hand side is not scoped and evaluates to `D`. The `x` in `Pattern[x, Blank[]]` is inside a function with the attribute `HoldFirst`. So it does not evaluate to `D`, but rather it is now the `x$`*number* variable created by `Module`. As a result, we have a definition of the form *f*[*x*_]=`D`. The pattern variable from the left-hand side does not appear on the right-hand side. Applying `Set` to `C[f[x$`*number*`_], D]` gives the definition *f*[*x*_]=`D`. After this definition is evaluated, `f[C]` finally yields `D`.

```
Module[{x = D, f}, Set @@ f[x_] ~ C ~ x; DownValues[f]]
```

As a result, we get `C - D`. The next input evaluates the original code fragment.

```
Module[{x = D, f}, C @@ f[x_] ~ Set ~ x // f[C]&] -
Module[{x = D, f}, Set @@ f[x_] ~ C ~ x // f[C]&]
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

n) Although all the elements of the first argument of `Union` are identical, the test applied by the `SameTest` option setting will always return `False`. Because `Union` with an explicit setting of the `SameTest` option carries out all needed comparisons (modulo transitivity), this means the first element has to be compared with 99 others, the second with 98 others, ... This makes $\sum_{k=1}^{99} k = 4950$ comparisons.

```
c = 0;
Union[Array[1&, {100}], SameTest -> ((c = c + 1; False)&)];
c
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

o) Obviously, on most computers, `virtualMatrix` cannot create a "real" matrix of size $10^6 \times 10^6$. The trick to generate such a matrix is to have only one "real" column and all other column being exactly identical. Here this is implemented.

```
virtualMatrix[dim_] :=
Module[{row = Table[1, {dim}]}, row /. 1 -> row]
```

$\mathcal{M}$ will now behave as a matrix.

```
M = virtualMatrix[10^6];
```

```
{MatrixQ[M], Dimensions[M], Length[M[[1]]],
        {M[[1, 1]], M[[-1, -1]]},
        M[[1000, 1000]] = 1000; M[[1000, 1000]]}
```

$\mathcal{M}$ also is a matrix, but not all elements are independently stored. So its actual memory usage is far smaller than for a "real" matrix.

```
{ByteCount[M], MemoryInUse[]}
```

Of course, operations that will make the rows different, or extract all elements (such as `Flatten[M]`) will need "real" memory and will very probably run out of memory. The following input changes one element per row. Now the rows become different and are stored as different entries. As a result, the real memory consumption increases.

```
Do[M[[k, 1]] = k; Print[MemoryInUse[]], {k, 5}]
```

For applications of such matrices, see, for instance, [281✱].

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

p) The `MapIndexed` function maps the pure function `(Part[`*expr*`, ##]& @@ #2)&` to the levels `{`*k*`, `*l*`}` of *expr*. The pure function depends only on the position of the part on which it acts. It extracts exactly the same part from *expr* that was there. As a result, *expr* itself is returned.

Here is an example expression.

```
expr = Log[x^2 + 5 x] + Sin[4 t^2 y^4] -
       (t y^(2 + Exp[-3 x])) + 45 t^6 - 4;
```

We use $-10 \le k, l \le 10$ and check that `MapIndexed[(Part[`*expr*`, ##]& @@ #2)&, `*expr*`, {`*k*`, `*l*`}, Heads -> True]` evaluates to the original expression.

```
Table[MapIndexed[(Part[expr, ##]& @@ #2)&,
                 expr, {k, l}, Heads -> True] === expr,
                 {k, -10, 10}, {l, -10, 10}] // Flatten // Union
```

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

24. Hash Value Collisions, Permutation Digit Sets

a) Experimenting suggests that the hash values have values in the order 10^9 $\left(\leq c = 2^{32}\right)$.

```
Hash /@ {2, Sqrt[3], E, N[Pi, 300], Sin[Catalan], x^x + Log[Sin[x]]}

N[%]
```

This means that when sampling about $\sqrt{c} = 2^{16} = 65\,536$ values, we expect to find two expressions hashed to the same hash value [271✱], [304✱]. (This is the idea of the birthday paradox used here [263✱], [10✱], [251✱].) So let us use hash 10^5 different large integers.

```
SeedRandom[111]

t = {Hash[#], #}& /@ Table[Random[Integer, {1, 10^15}], {10^5}];
```

We now have a few less than 100000 different hash values (the actual number depends on the computer system and the *Mathematica* session); this means we found some collisions. All randomly selected integers were different.)

```
{Length[Union[First /@ t]], Length[Union[Last /@ t]]}
```

Here are the pairs with the same hash value.

```
Map[Last, Select[Partition[Sort[t, (#1[[1]] < #2[[1]])&], 2, 1],
                 (#[[1, 1]] == #[[2, 1]])&], {2}]

Map[Hash, %, {2}]
```

We do not have to invoke `Random` here (we discuss `Random` in the Chapter 1 of the Graphics volume [301✱]). Trying to use the integers 1 to 10^5 will give 10^5 different hash values, but the numerical values of, say, $1/i$ for $1 \leq i \leq 10^5$ will be sometimes hashed to the same integer.

```
t = {Hash[N[#, 22]], #}& /@ Table[1/i, {i, 10^5}];
```

We now have less than 10000 different hash values; this means we found some collisions.

```
Length[Union[First /@ t]]

Map[Last, Select[Partition[Sort[t, (#1[[1]] < #2[[1]])&], 2, 1],
                 (#[[1, 1]] == #[[2, 1]])&], {2}]

Map[Hash, N[%, 22], {2}]
```

Be aware that the explicit hash values for the numerical approximations of $1/i$ depend on the precision used.

```
Map[Hash, N[%%, 30], {2}]
```

Hash values are operating system- and session-dependent.

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

b) To be general, we implement one function for the calculation of the set $\mathcal{M}_k^{(b)}$ that takes into account about efficiency, but does not take into account special properties of b and k (like divisibility rules based on the sums of the digits [247✱]).

There are various possible approaches to the calculation of the set $\mathcal{M}_o^{(b)}$. We could, for instance, loop over all k-digit integers and all multipliers m and select the pairs fulfilling the conditions on their digits. Here we choose a more time and memory efficient approach. We search for the s_1 and the multipliers m and build the digits of these numbers recursively from the end. Starting with an expression of the form `{{`*lastDigit*`}, {2,…,`$m_{\max}$`}}` we form the expressions `{{`*penultimateDigit*`, `*lastDigit*`}, {2,…,`$m_{\max}$`}}` and selects the multipliers m that are compatible with the conditions. Then we add the next digit and so on. Here $m_{\max}$ is the largest possible multiplier m, the integer part of the ratio of the largest to the smallest number from $\mathcal{S}_o^{(b)}$. The trailing digits *trailingDigits* are compatible with the multiplier m, if the last digits from the product are from $[1, k]$ and if the smallest and largest numbers having the trailing digits *trailingDigits* allows having the multiplier m. The two functions `nonRepeatingTrailingDigitsQ` and `minMax⋱ BoundQ` implement these two conditions.

```
(* no digits appears twice and come from [1, o] *)
nonRepeatingSequenceQ[l_, o_] := Length[Union[l]] === Length[l] &&
                                 Max[l] <= o && Min[l] =!= 0

(* the digits of the number n are fulfilling the conditions *)
nonRepeatingNumberQ[n_, k_, o_, base_] :=
            nonRepeatingSequenceQ[IntegerDigits[n, base, k], o]

(* the product of m and the number with trailing digits tDs
   is fulfilling the conditions *)
nonRepeatingTrailingDigitsQ[tDs:trailingDigits_, m_, o_, base_] :=
   nonRepeatingNumberQ[m FromDigits[tDs, base], Length[tDs], o, base]

(* the multiplier m is compatible with the trailing digits *)
minMaxBoundQ[tDs:trailingDigits_, m_, allDigits_, b:base_] :=
Block[{tDsM, minX, maxX, minY, maxY, sc, scM},
      tDsM = IntegerDigits[m FromDigits[tDs, base], base, Length[tDs]];
      {sc, scM} = Sort[Complement[allDigits, #]]& /@ {tDs, tDsM};
      (* smallest and largest number *)
      {minX,  maxX} = FromDigits[Join[#, tDs ], b]& /@ {sc , Reverse[sc ]};
      (* smallest and largest number after multiplication *)
      {minY,  maxY} = FromDigits[Join[#, tDsM], b]& /@ {scM, Reverse[scM]};
      (* bounds on the multiplier *)
      If[IntegerQ[#], IntegerPart[#], IntegerPart[#] + 1]&[minY/maxX] <=
         m <= IntegerPart[maxY/minX]]
```

Given an expression of the form `{{`*trailingDigits*`}, {`*possibleMultipliers*`}}`, the function `reduceMultiples` selects the possible multipliers from *possibleMultipliers*.

```
reduceMultiples[{tDs:trailingDigits_, m:possibleMultiples_},
                o_, allDigits_, base_] :=
{tDs, Select[m, (* apply the two conditions *)
                (nonRepeatingTrailingDigitsQ[tDs, #, o, base] &&
                 minMaxBoundQ[tDs, #, allDigits, base])&]}
```

To add a digit to the already present trailing digits and select the resulting possible sequences, we use the function `addDigit`.

```
addDigit[{tDs:trailingDigits_, m:possibleMultiples_},
         o_, allDigits_, base_] :=
   reduceMultiples[{#, m}, o, allDigits, base]& /@
                   (Join[{#}, tDs]& /@ Complement[allDigits, tDs])
```

The function `step` applies the function `addDigit` to a list of expressions and deletes the ones which have no possible multipliers.

```
step[tm:trailingDigitsAndMultiplesList_, o_, allDigits_, base_] :=
DeleteCases[Flatten[addDigit[#, o, allDigits, base]& /@ tm, 1], {_, {}}]
```

Putting now all these functions together, we arrive at the function `findDigitsAndMultiples`.

```
findDigitsAndMultiples[o_, base_] :=
Module[{allDigits = Range[o], start},
       (* fill in all first digits and all multipliers *)
       start = {{#}, Range[2,
                      IntegerPart[FromDigits[Reverse @ allDigits, base]/
                      FromDigits[allDigits, base]]]}& /@ allDigits;
       (* add all o - 1 remaining digits *)
       Nest[step[#, o, allDigits, base]&, start, o - 1]]
```

Here is the simplest example: $\mathcal{M}_3^{(4)} = \{\{123_4, 312_4\}\}$.

```
findDigitsAndMultiples[3, 4]
```

To format the results nicely, we implement a function `formatIdentities`.

```
formatIdentities[{digits_, multipliers_}, base_] :=
Block[{Equal, Times}, (HoldForm @@
{Equal[Times[BaseForm[#, base], BaseForm[FromDigits[digits, base], base]],
       BaseForm[# FromDigits[digits, base], base]]})& /@ multipliers]
```

Here are the seven pairs from the set $\mathcal{M}_6^{(7)}$.

```
formatIdentities[#, 7]& /@ findDigitsAndMultiples[6, 7]
```

The smallest o yielding nontrivial solutions in base 10 is $o = 8$. The 2270 different s_1 are calculated in a few seconds.

```
Timing[Length[fdsm810 = findDigitsAndMultiples[8, 10]]]
```

Here are the solutions from the last set that have the largest multiplicity (three).

```
formatIdentities[#, 10]& /@
Function[λ, Select[fdsm810, Length[Last[#]] == λ&]][
        (* largest multiplicity *) Max[Length[Last[#]]& /@ fdsm810]]
```

And here is the number of pairs of the sets $\mathcal{M}_o^{(b)}$ for $2 \le b \le 10$, $1 \le k \le b-1$. The base b increases downwards and o increases horizontally.

```
With[{bMax = 10}, TableForm[
      Table[If[j < b, (* add multiplicity *)
       Plus @@ (Length[Last[#]]& /@
                               findDigitsAndMultiples[j, b]), "-"],
         {b, 2, bMax}, {j, bMax - 1}], TableSpacing -> 1,
         TableHeadings -> {Range[2, bMax], Range[1, bMax - 1]},
         TableAlignments -> Center]]
```

Now it is straightforward to calculate the cardinality of $\mathcal{M}_{11}^{(12)}$ using `numberOfSolution[findDigitsAnd`⋱ `Multiples[11, 12]]`. The result is 2017603.

```
Σ (* session summary *) TMGBs`PrintSessionSummary[]
```

25. Function Calls in `GluedPolygons`

This was the code for the construction of the glued polygons.

```
GluedPolygons[n_Integer?(# >= 3&), angle:α_?(Im[N[#]] === 0&),
              iter__Integer?(# >= 0&), faceShape:(Polygon | Line),
              opts___Rule] :=
Module[{c = N[Cos[α]], s = N[Sin[α]], myUnion, r, ℛ, allm, argch,
        makeHole, makeLine, n = #/Sqrt[#.#]&, ε = 10^-6},
(* a completely transitive Union  *)
myUnion[l_] := Union[l, SameTest -> ((Plus @@ (#.#& /@ (#1 - #2))) < ε&)];
(* construction of next layer *)
(* rotate a point *)
r[point_, rotPoint_, {dir1_, dir2_, dir3_}] :=
 Module[{δ = point - rotPoint, parallel, normal},
        parallel = δ.dir1 dir1;
        normal = Sqrt[#.#]&[δ - parallel];
        rotPoint + c normal dir2 + s normal dir3 + parallel];
(* rotate points *)
ℛ[l_] :=
Module[{dir1, dir2, dir3, first},
       (* 3 orthogonal directions *)
       dir1 = n[Subtract @@ Take[l, 2]];
       dir2 = n[(Plus @@ l)/Length[l] - (Plus @@ Take[l, 2])/2];
       dir3 = -Cross[dir1, dir2];
       Map[N[r[#, l[[1]], {dir1, dir2, dir3}]]&, l, {-2}]];
(* prepare lists *)
allm[l_] := Table[RotateLeft[l, i], {i, Length[l] - 1}];
argch[l_] := Join[Reverse[Take[l, 2]], Reverse[Drop[l, 2]]];
(* make a hole in a polygon *)
makeHole[l_] :=
 With[{mp = (Plus @@ l)/Length[l], h = Append[#, First[#]]&[l]},
       MapThread[Polygon[Join[#1, Reverse[#2]]]&,
      {Partition[h, 2, 1], Partition[mp + 0.8(# - mp)& /@ h, 2, 1]}]];
(* wireframe or polygons *)
makeLine[l_] := Line[Append[l, First[l]]];
(* show graphics *)
Show[Graphics3D[If[faceShape === Polygon, makeHole[#], makeLine[#]]& /@
 Join[{Table[N[{Cos[φ], Sin[φ], 0}], {φ, 0, 2Pi - 2Pi/n, 2Pi/n}]},
(* build layer on layer *)
If[iter > 0, Flatten[NestList[myUnion[argch /@ (ℛ /@
 Flatten[Join[allm /@ #], 1])]&, Join[argch /@ (ℛ /@ #)]&[(* one face *)
    Table[Table[N[{Cos[φ], Sin[φ], 0}], {φ, φ0, φ0 + 2Pi - 2Pi/n, 2Pi/n}],
          {φ0, 0, 2Pi - 2Pi/n, 2Pi/n}]], iter - 1], 1], {}]]], opts]]
```

These are the functions we are interested in.

```
interestingFunctions =  {Reverse, Join, Dot, Map, Partition,
          Apply, Take, MapThread, Drop, Table, Part, Flatten};
```

To monitor the number of calls to the built-in function *func*, we unprotect these functions and add a new rule to it. The new rule never matches (the `False` in the condition), but as a side effect of the test, we monitor that they were called.

```
(Unprotect[#]; counter[#] = 0;
 HoldPattern[#[___]] := Null /; (counter[#] = counter[#] + 1; False))& /@
                                          interestingFunctions;
```

Now, we run the construction of the glued polygons.

```
GluedPolygons[5, 3Pi/4, 1, Polygon, DisplayFunction -> Identity];
```

Here is the actual number of calls to the functions under consideration.

```
{#, counter[#]}& /@ interestingFunctions
```

As a side effect in the condition testing, we not only monitor the call itself, but we also store the arguments used to call *func*. Here, this is implemented.

```
(Unprotect[#]; bag[#] = Bag[];
 HoldPattern[#[args___]] := Null /;
              (bag[#] = Bag[bag[#], Bag[args]]; False))& /@
                                        interestingFunctions;
```

Now, we run the construction of the glued polygons again.

```
GluedPolygons[5, 3Pi/4, 1, Polygon, DisplayFunction -> Identity];
```

For instance, `Apply` was called 16 times with `Plus` as its first argument.

```
Count[bag[Apply], Plus, Infinity]
```

Σ (* session summary *) `TMGBs`PrintSessionSummary[]`

References

✱1 P. Abbott. *The Mathematica Journal* 3, n1 (1992).

✱2 L. Aceto, D. Trigiante. *Rend. Circ. Mat. Palermo* S 68, 219 (2002).

✱3 A. Adler. *Math. Intell.* 14, n3, 14 (1992).

✱4 A. Adler, L. C. Washington. *J. Number Th.* 52, 179 (1995). *DOI-Link*

✱5 Y. Aharonov, L. Davidovich, N. Zagury. *Phys. Rev.* A 48, 1687 (1993). *DOI-Link*

✱6 M. Ahmed, J. De Loera, R. Hemmecke. *arXiv:math.CO*/0201108 (2002). *Get Preprint*

✱7 R. Albert. A.-L. Barabási. *arXiv:cond-mat*/0106096 (2001). *Get Preprint*

✱8 R. Aldrovandi. *Special Matrices of Mathematical Physics*, World Scientific, Singapore, 2001. *BookLink*

✱9 L. Alexander, R. Johnson, J. Weiss. *1998 Proc. Stat. Edu.*, American Statistical Association, Alexandria, 1998.

✱10 V. Ambegaokar. *Reasoning about Luck*, Cambridge University Press, Cambridge, 1996. *BookLink (2)*

✱11 O. D. Anderson. *Int. J. Math. Edu. Sci. Technol.* 23, 131 (1992).

✱12 M. E. Andersson. *Acta Arithm.* 85, 301 (1998).

✱13 M. Andrecut, M. K. Ali. *Phys. Lett.* A 326, 328 (2004). *DOI-Link*

✱14 D. F. Andrews, A. M. Herzberg. *Data*, Springer-Verlag, New York, 1984. *BookLink*

✱15 W. S. Andrews. *Magic Squares and Cubes*, Open Court, Chicago, 1908. *BookLink (2)*

✱16 L. J. Anthony, H. East, M. J. Slater. *Rep. Progr. Phys.* 32, 709 (1969). *DOI-Link*

✱17 T. M. Apostol. *Am. Math. Monthly* 76, 289 (1969).

✶18 A. Arache. *Am. Math. Monthly* 72, 861 (1965).

✶19 V. I. Arnold, A. Avez. *Ergodic Problems of Classical Mechanics*, Benjamin, New York, 1968. *BookLink*

✶20 Y. Avishai, D. Berend. *J. Phys.* A 26, 2437 (1993). *DOI-Link*

✶21 M. Ayala–Sánchez. *arXiv:physics*/0208068 (2002). *Get Preprint*

✶22 N. B. Backhouse, A. G. Fellouris. *J. Phys.* A 17, 1389 (1984). *DOI-Link*

✶23 H. F. Bauch. *Math. Semesterber.* 38, 99 (1991).

✶24 D. H. Bailey, J. M. Borwein, P. B. Borwein, S. Plouffe. *Math. Intell.* 19, 590 (1997).

✶25 S. Barnett. *Matrices,* Clarendon Press, Oxford, 1990. *BookLink (2)*

✶26 R. Bass. *J. Math. Phys.* 26, 3068 (1985). *DOI-Link*

✶27 J. Baylis. *Math. Gaz.* 69, 95 (1985).

✶28 M. Beck, M. Cohen, J. Cuomo, P. Gribelyuk. *Am. Math. Monthly* 110, 707 (2003).

✶29 R. Becker, F. Sauter. *Theorie der Elektrizität*, Teubner, Stuttgart, 1962. *BookLink (2)*

✶30 D. W. Belousek, E. B. Flint, J. P. Kenny, K. R. Roos. *Chaos, Solitons, Fractals* 7, 853 (1996). *DOI-Link*

✶31 D. Belov, A. Konechny. *arXiv:hep-th*/0210169 (2002). *Get Preprint*

✶32 F. Benford. *Proc. Am. Philos. Soc.* 78, 551 (1938).

✶33 A. Ben–Israel, T. N. E. Greville. *Generalized Inverses*, Springer-Verlag, New York, 2003. *BookLink*

✶34 W. H. Benson, O. Jacoby. *New Recreations with Magic Squares*, Dover, New York, 1976. *BookLink*

✶35 H. Bergold. *Didaktik Math.* 4, 266 (1979).

✶36 B. C. Berndt. *Ramanujan*'*s Notebooks,* Part I, Springer-Verlag, New York, 1985. *BookLink*

✶37 M. Bicknell, V. Hoggatt. *Recr. Math. Mag.* n13, 13 (1964).

✶38 D. Birmajer. *arXiv:math.RA*/0305430 (2003). *Get Preprint*

✶39 L. Blanchet, G. Faye. *J. Math. Phys.* 42, 4391 (2001). *DOI-Link*

✶40 M. Bóna. *Studies Appl. Math.* 94, 415 (1995).

✶41 A. L. Bondarev. *Teor. Mat. Fiz.* 101, 315 (1994).

✶42 V. I. Borodulin, R. N. Rogalyov, S. R. Slabospitsky. *arXiv:hep-ph*/9507456 (1995). *Get Preprint*

✶43 A. Brauer. *Am. Math. Monthly* 53, 521 (1946).

✱44 A. Bremner. *Acta Arithm.* 88, 289 (1999).

✱45 G. K. Brennen, J. E. Williams. *arXiv:quant-ph*/0306056 (2003). *Get Preprint*

✱46 R. Brown, J. L. Merzel. *Period. Math. Hung.* 47, 45 (2003). *DOI-Link*

✱47 R. B. Brown, A. Gray. *Comm. Math. Helv.* 42, 222 (1967).

✱48 M. Brückner. *Acta Leopold.* 86, 1, (1906).

✱49 R. C. Brunet. *J. Math. Phys.* 16, 1112 (1975). *DOI-Link*

✱50 B. Buck, A. C. Merchant, S. M. Perez. *Eur. J. Phys.* 14, 59 (1993). *DOI-Link*

✱51 J. Burke, E. Kincanon. *Am. J. Phys.* 59, 952 (1991). *DOI-Link*

✱52 F. Calgaro. *J. Comput. Appl. Math.* 83, 127 (1997). *DOI-Link*

✱53 F. Calogero, A. M. Perelomov. *arXiv:math-ph*/0112014 (2001). *Get Preprint*

✱54 F. Calogero. *Classical Many-Body Problems Amenable to Exact Treatments*, Springer-Verlag, Berlin, 2001. *BookLink*

✱55 A. L. Candy. *Construction Classification and Census of Magic Squares of an Even Order*, Edwards Brothers, Ann Arbor, 1937. *BookLink*

✱56 R. Carbó, E. Besalu. *Comput. Chem.* 18, 117 (1994).

✱57 D. E. Carlson, A. Hoger. *Quart. J. Appl. Math.* 44, 409 (1986).

✱58 B. W. Char in A. Griewank, G. F. Corliss (eds.). *Automatic Differentiation of Algorithms: Theory, Implementa tion, and Application*, SIAM, Philadelphia, 1991. *BookLink*

✱59 C. A. Charalambides, J. Singh. *Commun. Stat.-Theor. Methods* 17, 2533 (1988).

✱60 F. Chung, S.-T. Yau. *J. Combinat. Th.* A 91, 191 (2000). *DOI-Link*

✱61 D. I. A. Cohen. *J. Combinat. Th.* A 20, 367 (1976).

✱62 R. K. Cooper, C. Pellegrini. *Modern Analytic Mechanic*, Kluwer, New York, 1999. *BookLink*

✱63 D. R. Curtiss. *Bull. Am. Math. Soc.* 17, 463 (1911).

✱64 M. Daumer, D. Dürr, S. Goldstein, N. Zanghí. *arXiv:quant-ph*/9601013 (1996). *Get Preprint*

✱65 J. A. Davies. *Eur. J. Phys.* B 27, 445 (2002). *DOI-Link*

✱66 M. A. B. Deakin. *Austral. Math. Soc. Gaz.* 20, 149 (1993).

✱67 E. Defez, L. Jódar. *J. Comput. Appl. Math.* 99, 105 (1998). *DOI-Link*

✱68 F. M. Dekking in R. V. Moody (ed.). *The Mathematics of Long-Range Aperiodic Order*, Kluwer, Dordrecht, 1997.

BookLink

✱69 E. A. de Souza Neto. *Comput. Meth. Appl. Mech. Eng.* 190, 2377 (2001). *DOI-Link*

✱70 E. A. de Souza Neto. *Int. J. Numer. Meth. Eng.* 61, 880 (2004). *DOI-Link*

✱71 P. Diaconis. *Ann. Prob.* 5, 72 (1977).

✱72 L. E. Dickson. *History of the Theory of Numbers*, v. I, Chelsea, New York, 1952. *BookLink (3)*

✱73 Y. I. Dimitrienko. *Tensor Analysis and Nonlinear Tensor Functions*, Kluwer, Dordrecht, 2002. *BookLink*

✱74 A. Dittmer. *Am. Math. Monthly* 101, 887 (1994).

✱75 A. P. Domoryad. *Mathematical Games and Pastimes*, MacMillan, New York, 1964. *BookLink*

✱76 S. N. Dorogovtsev, J. F. F. Mendes. *arXiv:cond-mat*/0105093 (2001). *Get Preprint*

✱77 S. Dubuc in R. Liedl, L. Reich, G. Targonski (eds.). *Iteration Theory and its Functional Equations*, Springer-Verlag, Berlin, 1985. *BookLink*

✱78 U. Dudley. *Mathematical Cranks*, Mathematical Association of America, Washington, 1992. *BookLink*

✱79 D. Dumont. *Math. Comput.* 33, 1293 (1979).

✱80 R. V. Durand, C. Franck. *J. Phys.* A 32, 4955 (1999). *DOI-Link*

✱81 M. Dvornikov. *arXiv:hep-ph*/0411101 (2004). *Get Preprint*

✱82 W. Ebeling, T. Pöshel. *Europhys. Lett.* 26, 241 (1994).

✱83 A. Edalat, P. J. Potts. *Electr. Notes Theor. Comput. Sc.* 6, (1997). http://www.elsevier.com/gej-ng/31/29/23/31/23/61/tcs6007.ps

✱84 S. B. Edgar, A. Höglund. *arXiv:gr-qc*/0105066 (2001). *Get Preprint*

✱85 E. Elizalde. *arXiv:cond-mat*/9906229 (1999). *Get Preprint*

✱86 P. Erdős, V. Lev, G. Rauzy, C. Sandor, A. Sárközy. *Discr. Math.* 200, 119 (1999). *DOI-Link*

✱87 A. G. Fellouris, L. K. Matiadou. *J. Phys.* A 35, 9183 (2002). *DOI-Link*

✱88 E. Fick. *Einführung in die Grundlagen der Quantenmechanik*, Geest and Portig, Leipzig, 1981. *BookLink*

✱89 M. Fiedler. *Lin. Alg. Appl.* 372, 325 (2003). *DOI-Link*

✱90 E. Formanek. *J. Algebra* 258, 310 (2002). *DOI-Link*

✱91 M. Friedman, A. Kandel. *Fundamentals of Computer Analysis*, CRC Press, Boca Raton, 1994. *BookLink*

✱92 C.-E. Froeberg. *Numerical Mathematics*, Addison-Wesley, Redwood City, 1985. *BookLink*

✱93 B. R. Frieden. *Found. Phys.* 9, 883 (1986). *DOI-Link*

✶94 L. V. Furlan. *Das Harmoniegesetz der Statistik*, Verlag für Recht und Gesellschaft AG, Basel, 1946. *BookLink*

✶95 S. Fussy, G. Grössing, H. Schwabl, A. Scrinzi. *Phys. Rev.* A 48, 3470 (1993). *DOI-Link*

✶96 P. Gaillard, V. Matveev. *Preprints MPI* 31-2002 (2002). http://www.mpim-bonn.mpg.de/cgi-bin/preprint/preprint_search.pl/MPI-2002-31.ps?ps=MPI-2002-31

✶97 S. Galam. *Physica* A 274, 132 (1999). *DOI-Link*

✶98 F. R. Gantmacher. *The Theory of Matrices*, Chelsea, New York, 1959. *BookLink (5)*

✶99 S. Garfunkel, C. A. Steen. *Mathematik in der Praxis*, Spektrum der Wissenschaft-Verlag, Heidelberg, 1989. *BookLink*

✶100 R. S. Garibaldi. *arXiv:math.LA*/0203276 (2002). *Get Preprint*

✶101 I. Gelfand, S. Gelfand, V. Retakh, R. Wilson. *arXiv:math.QA*/0208146 (2002). *Get Preprint*

✶102 D. V. Georgievskiĭ, M. V. Shamolin. *Dokl. Phys.* 380, 47 (2001).

✶103 H. Gies. *arXiv:hep-th*/9909500 (1999). *Get Preprint*

✶104 F. Gobel, R. P. Nederpelt. *Am. Math. Monthly* 78, 1097 (1971).

✶105 V. V. Goldman, J. H. J. Molenkamp, J. A. van Hulzen in A. Griewank, G. F. Corliss (eds.). *Automatic Differentia tion of Algorithms: Theory, Implementation, and Application*, SIAM, Philadelphia, 1991. *BookLink*

✶106 R. N. Goldman in D. Kirk (ed.). *Graphics Gems* III, Academic Press, Boston, 1992. *BookLink*

✶107 R. Goldman. *IEEE Comput. Graphics Appl.* n3, 66 (2003).

✶108 E. Goles, M. Morvan, H. D. Phan in D. Krob, A. A. Mikhalev, A. V. Mikhalev (eds.). *Formal Power Series and Algebraic Combinatorics*, Springer-Verlag, Berlin, 2000. *BookLink*

✶109 G. H. Golub, C. F. van Loan. *Matrix Computations*, Johns Hopkins University Press, Baltimore, 1989. *BookLink (4)*

✶110 L. L. Gonçalves, L. B. Gonçalves. *arXiv:cond-mat*/0501136 (2005). *Get Preprint*

✶111 G. A. Gottwald, M. Nicol. *Physica* A 303, 387 (2002). *DOI-Link*

✶112 A. Graham. *Kronecker Products and Matrix Calculus: with Applications*, Ellis Horwood, Chichester, 1981. *BookLink (2)*

✶113 F. Graner in B. Dubrulle, F. Graner, D. Sornette (eds.). *Scale Invariance and Beyond* Springer-Verlag, Berlin, 1997. *BookLink*

✶114 F. A. Graybill. *Introduction to Matrices with Applications in Statistics*, Wadsworth, Belmont, 1969. *BookLink*

✶115 M. Gross, A. Hubeli. *Preprint* ETH 338/2000 (2000). ftp://ftp.inf.ethz.ch/pub/publications/tech-reports/3xx/338.abstract

✱116 G. Grössing. *Phys. Lett. A* 131, 1 (1988). *DOI-Link*

✱117 G. Grössing. *Physica D* 50, 321 (1991). *DOI-Link*

✱118 G. Grössing, A. Zeilinger. *Physica* B+C 151, 366 (1988).

✱119 G. Grössing, A. Zeilinger. *Physica* D 31, 70 (1988). *DOI-Link*

✱120 M. G. Guillemot. *Europhys. Lett.* 53, 155, (2001). *DOI-Link*

✱121 H. Guiter, M. V. Arapov. *Studies on Zipf's law*, Studienverlag Dr. N. Brockmeyer, Bochum, 1982. *BookLink*

✱122 R. K. Guy, J. F. Selfridge. *Am. Math. Monthly* 80, 868 (1973).

✱123 K. B. Hajra, P.Sen. *arXiv:cond-mat*/0409017 (2004). *Get Preprint*

✱124 M. Halibard, I. Kanter. *Physica* A 249, 525 (1998). *DOI-Link*

✱125 A. J. Hanson in P. S. Heckbert (ed.). *Graphics Gems* IV, Academic Press, Boston, 1994. *BookLink*

✱126 J. F. Harris. *Ph. D. Thesis*, Canterbury, 1999.

✱127 W. A. Harris, Jr., J. P. Fillmore, D. R. Smith. *SIAM Rev.* 43, 694 (2001). *DOI-Link*

✱128 R. Haydock. *J. Phys.* A 7, 2120 (1974). *DOI-Link*

✱129 R. Heckmann. *Theor. Comput. Sc.* 279, 65, (2002). *DOI-Link*

✱130 E. R. Hedrick. *Ann. Math.* 1, 49 (1899).

✱131 C. J. Henrich. *Am. Math. Monthly* 98, 481 (1991).

✱132 H. Hemme. *Bild der Wissenschaft* n11, 178 (1987).

✱133 H. Hemme. *Bild der Wissenschaft* n10, 164 (1988).

✱134 H. Hemme. *Bild der Wissenschaft* n9, 143 (1989).

✱135 E. Herlt, N. Salié. *Spezielle Relativitätstheorie*, Akademie-Verlag, Berlin, 1978. *BookLink*

✱136 N. J. Highham. *Math. Comput.* 46, 537 (1986).

✱137 T. P. Hill. *Am. Math. Monthly* 102, 322 (1995).

✱138 T. P. Hill. *Proc. Am. Math. Soc.* 123, 887 (1995).

✱139 T. P. Hill. *Stat. Sci.* 10, 354 (1995).

✱140 D. E. Holz, H. Orland, A. Zee. *arXiv:math-ph*/0204015 (2002). *Get Preprint*

✱141 R. Honsberger. *Ingenuity in Mathematics*, Random House, New York, 1970. *BookLink*

★142 R. Honsberger. *More Mathematical Morsels*, American Mathematical Society, 1991. *BookLink (2)*

★143 S. Humphries, C. Krattenthaler. *arXiv:math.AC*/0411061 (2004). *Get Preprint*

★144 J. A. H. Hunter, J. S. Madachy. *Mathematical Diversions*, Van Nostrand, Princeton, 1963. *BookLink*

★145 W. Hürlimann. *MPS: Pure mathematics*/0306006 (2003).
http://www.mathpreprints.com/math/Preprint/werner.huerlimann/20030603/1/IntPowers.pdf

★146 W. Hürlimann. *MPS: Pure mathematics*/0306013 (2003).
http://www.mathpreprints.com/math/Preprint/werner.huerlimann/20030624/1/GenBenford.pdf

★147 A. Ilachinski. *Cellular Automata*, World Scientific, Singapore, 2001. *BookLink*

★148 D. Ismailescu, R. Radoičić. *Comput. Geom.* 27, 257 (2004). *DOI-Link*

★149 M. Itskov. *ZAMM* 82, 535 (2002). *DOI-Link*

★150 D. M. Jackson, R. Aleliunas. *Can. J. Math.* 29, 971 (1977).

★151 B. C. Johnson. *Methol. Comput. Appl. Prob.* 3, 35 (2001). *DOI-Link*

★152 J.-M. Jolion. *J. Math. Imag. Vision* 14, 73 (2002). *DOI-Link*

★153 B. K. Jones in D. Abbott, L. B. Kish (eds.). *Unsolved Problems of Noise and Fluctuations*, American Institute of Physics, Melville, 2000. *BookLink*

★154 J. H. Jordan. *Am. Math. Monthly* 71, 61 (1964).

★155 S. A. Kamal. *Matrix Tensors Quart.* 31, 64 (1981).

★156 I. Kantner, D. A. Kessler. *Phys. Rev. Lett.* 74, 4559 (1995). *DOI-Link*

★157 Y. Kawamura. *Progr. Theor. Phys.* 107, 1105 (2002).

★158 J. D. O'Keeffe. *Int. J. Math. Edu. Sci. Technol.* 12, 541 (1981).

★159 J. S. Kelly. *Arrow Impossibility Theorems*, Academic Press, New York, 1978. *BookLink*

★160 R. Kerner. *arXiv:math-ph*/0011023 (2000). *Get Preprint*

★161 I. Kim, G. Mahler. *arXiv:quant-ph*/9902020 (1999). *Get Preprint*

★162 I. Kim, G. Mahler. *arXiv:quant-ph*/9902024 (1999). *Get Preprint*

★163 J. B. Kim, J. E. Dowdy. *J. Korean Math. Soc.* 17, 141 (1980).

★164 D. E. Knuth. *The Art of Computer Programming*, v.2, Addison-Wesley, Reading, 1969. *BookLink*

★165 D. E. Knuth. *The Art of Computer Programming*, v. 3, Addison-Wesley, Reading, 1998. *BookLink*

★166 I. Kogan, A. M. Perelomov, G. W. Semenoff. *arXiv:math-ph*/0205038 (2002). *Get Preprint*

✱167 W. Kolakoski. *Am. Math. Monthly* 72, 674 (1965).

✱168 A. V. Kontorovich, S. J. Miller. *arXiv:math.NT*/0412003 (2004). *Get Preprint*

✱169 K. Kopferman. *Mathematische Aspekte der Wahlverfahren*, BI, Mannheim, 1991. *BookLink*

✱170 Y. N. Kosovtsov. *arXiv:math-ph*/0409035 (2004). *Get Preprint*

✱171 G. Kowalewski. *Magische Quadrate und magische Parkette*, Teubner, Leipzig, 1939.

✱172 A. Kozlowski. *The Mathematica Journal* 9, 483 (2004).

✱173 M. Kraitchik. *Mathematical Recreations*, Dover, New York, 1953. *BookLink*

✱174 C. Krattenthaler. *Sém. Lothar. Combinat.* B 42q (1999). http://80-www.mat.univie.ac.at.proxy2.library.uiuc.edu/~slc/wpapers/s42kratt.html

✱175 C. Krattenthaler. *arXiv:math.CO*/0503507 (2005). *Get Preprint*

✱176 W. A. Kreiner. *Z. Naturf.* 58a, 618 (2003).

✱177 F. D. Kronewitter. *arXiv:math.LA*/0101245 (2001). *Get Preprint*

✱178 H. Kučera, W. N. Francis. *Computational Analysis of Present-Day American English*, Brown University Press, Providence, 1970. *BookLink*

✱179 S. Kunoff. *Fibon. Quart.* 25, 365 (1987).

✱180 S. Lakić, M. S. Petković. *ZAMM* 78, 173 (1998). *DOI-Link*

✱181 A. Lakshminarajan, N. L. Balazs. *Ann. Phys.* 226, 350 (1993). *DOI-Link*

✱182 P. Lancaster, M. Tismenetsky. *The Theory of Matrices*, Academic Press, Orlando, 1985. *BookLink*

✱183 C. T. Lang. *Fibon. Quart.* 24, 349 (1986).

✱184 A. Lascoux. *Ann. Combinat.* 1, 91 (1997).

✱185 D. H. Lehmer. *Am. Math. Monthly* 37, 294 (1930).

✱186 D. S. Lemons. *Am. J. Phys.* 54, 816 (1986). *DOI-Link*

✱187 D. Lenares. *Proc. ACRL 1999* (1999). http://www.ala.org/acrl/lenares.pdf

✱188 I. E. Leonard. *SIAM Rev.* 38, 507 (1996). *DOI-Link*

✱189 Y. L. Loh, S. N. Taraskin, S. R. Elliott. *Phys. Rev.* E 63, 056706 (2001). *DOI-Link*

✱190 M. Lotan. *Am. Math. Monthly* 56, 535 (1948).

✱191 T.-T. Lu, S.-H. Shiou. *Comput. Math. Appl.* 43, 119 (2002). *DOI-Link*

✱192 H. Lütkepohl. *Handbook of Matrices*, John Wiley, Chichester, 1996. *BookLink (2)*

✱193 I. Marek, K. Zitny. *Matrix Analysis for Applied Sciences I*, Teubner, Stuttgart, 1983.

✱194 R. Maeder. *Programming in Mathematica*, Addison-Wesley, Reading, 1991. *BookLink (3)*

✱195 R. Maeder. *The Mathematica Journal* 2, n1, 37 (1992).

✱196 H. M. Mahmoud. *Sorting*, Wiley, New York, 2000. *BookLink*

✱197 L. C. Malacarne, R. S. Mendes. *Physica* A 286, 391 (2000). *DOI-Link*

✱198 B. J. Malešević. *Univ. Beograd Publ. Elektrotehn. Fak.* 7, 105 (1998).

✱199 B. J. Malešević. *Univ. Beograd Publ. Elektrotehn. Fak.* 9, 29 (1998).

✱200 B. J. Malešević. *arXiv:math.CO*/0409287 (2004). *Get Preprint*

✱201 M. Marsili, Y.-C. Zhang. *Phys. Rev. Lett.* 80, 2741 (1998). *DOI-Link*

✱202 H. Martini in T. Bisztriczky, P. McMullen, R. Schneider, A, Ivić Weiss. *Polytopes: Abstract, Convex and Computational*, Kluwer, Dordrecht, 1994. *BookLink*

✱203 H. Martini in O. Giering, J. Hoschek (eds.). *Geometrie und ihre Anwendungen*, Carl Hanser, München, 1994. *BookLink*

✱204 G. Másson, B. Shapiro. *Exper. Math.* 10, 609 (2001).

✱205 C. Mauduit in J.-M. Gambaudo, P. Hubert, P. Tisseur, S. Vaienti (eds.). *Dynamical Systems*, World Scientific, Singapore, 2000. *BookLink*

✱206 B. M. McCoy. *Int. J. Mod. Phys.* A 14, 3921 (1999). *DOI-Link*

✱207 D. P. Mehendale. *arXiv:math.GM*/0503578 (2005). *Get Preprint*

✱208 E. Meissel. *Math. Ann.* 2, 636 (1870).

✱209 E. Meissel. *Math. Ann.* 3, 523 (1870).

✱210 D. A. Meyer. *arXiv:quant-ph*/0111069 (2001). *Get Preprint*

✱211 D. Middleton. *An Introduction to Statistical Communication Theory*, McGraw–Hill, New York, 1960. *BookLink*

✱212 R. Miller. *Am. Math. Monthly* 85, 183 (1978).

✱213 R. Milson. *arXiv:math.CO*/0003126 (2000). *Get Preprint*

✱214 A. Miyake. *arXiv:quant-ph*/0206111 (2002). *Get Preprint*

✱215 A. Miyake, M. Wadati. *arXiv:quant-ph*/0212146 (2002). *Get Preprint*

✱216 M. A. Montemurro. *Physica* A 300, 567 (2001). *DOI-Link*

✱217 M. A. Montemurro in M. Gell-Mann, C. Tsallis. *Nonextensive Entropy–Interdisciplinary Applications*, Oxford University Press, Oxford, 2004. *BookLink (2)*

✱218 A. Moesner. Sitzungsberichte *Math.-Naturw. Klasse der Bayerischen Akademie der Wissenschaften* 29, 1952 (1951).

✱219 C. Moler, C. Van Loan. *SIAM Rev.* 45, 3 (2003). *DOI-Link*

✱220 H. Moritz, B. Hofmann-Wellenhof. *Geometry, Relativity, Geodesy*, Whichmann, Karlsruhe, 1993. *BookLink*

✱221 T. Muir. *A Treatise on the Theory of Determinants*, Dover, New York 1960. *BookLink (2)*

✱222 G. L. Naber. *The Geometry of Minkowski Spacetime*, Springer-Verlag, New York, 1992. *BookLink (2)*

✱223 W. Narkiewicz. *The Development of Prime Number Theory*, Springer-Verlag, Berlin, 2000. *BookLink*

✱224 A. Nayak, A. Vishwanath. *arXiv:quant-ph*/0010117 (2000). *Get Preprint*

✱225 M. E. J. Newman. *arXiv:cond-mat*/0011144 (2000). *Get Preprint*

✱226 M. E. J. Newman. *Proc. Natl. Acad. Sci. USA* 98, 404 (2001). *DOI-Link*

✱227 M. E. J. Newman. *arXiv:cond-mat*/0412004 (2004). *Get Preprint*

✱228 M. J. Nigrini. *J. Am. Tax Ass.* 18, 72 (1996).

✱229 T. Nowicki. *Invent. Math.* 144, 233 (2001). *DOI-Link*

✱230 A. Odlyzko. *Preprint* (2000). http://www.research.att.com/~amo/doc/rapid.evolution.abst

✱231 R. Oldenburger. *Am. Math. Monthly* 47, 25 (1940).

✱232 I. Paasche. *Compositio Math.* 12, 263 (1956).

✱233 A. Palazzolo. *Am. J. Phys.* 44, 63 (1976). *DOI-Link*

✱234 F. Palmer (ed.). *Selected Papers by J. R. Firth*, Longman, London, 1968. *BookLink*

✱235 B. N. Parlett. *Lin. Alg. Appl.* 355, 85 (2002). *DOI-Link*

✱236 E. Pascal. *Die Determinanten*, Teubner, Leipzig, 1900.

✱237 W. Pauli. *Theory of Relativity*, Pergamon Press, New York, 1958. *BookLink*

✱238 M. Peczarski in R. Möhring, R. Raman (eds.). *Algorithms - ESA 2002*, Springer-Verlag, Berlin, 2002. *BookLink*

✱239 A. R. Penner. *Am. J. Phys.* 69, 332, (2001). *DOI-Link*

✱240 R. Perline. *Phys. Rev.* E 54, 220 (1996). *DOI-Link*

✱241 L. Pietronero, E. Tosatti, V. Tosatti, A. Vespignani. *arXiv:cond-mat* 9808305 (1998). *Get Preprint*

✱242 L. Pietronero, E. Tosatti, V. Tosatti, A. Vespignani. *Physica* A 293, 297 (2001). *DOI-Link*

✱243 R. S. Pinkham. *Ann. Math. Stat.* 32, 1223 (1962).

✱244 J. F. Plebanski, M. Przanowski. *J. Math. Phys.* 29, 2334 (1988). *DOI-Link*

✱245 E. R. Prakasan, A. Kumar, A. Sagar, L. Mohan, S. K. Singh, V. L. Kalyane, V. Kumar. *arXiv:physics*/0308107 (2003). *Get Preprint*

✱246 D. Prato, C. Tsallis. *J. Math. Phys.* 41, 3278 (2000). *DOI-Link*

✱247 J.-C. Puchta, J. Spilker. *Math. Semesterber.* 49, 209 (2002). *DOI-Link*

✱248 E. J. Putzer. *Am. J. Math.* 73, 2 (1966).

✱249 R. A. Raimi. *Am. Math. Monthly* 83, 521 (1976).

✱250 L. Rastelli, A. Sen, B. Zwiebach. *arXiv:hep-th*/0111281 (2001). *Get Preprint*

✱251 P. N. Rathie, P. Zörnig. *Int. J. Math. Math. Sci.* 60, 3827 (2003).

✱252 P. Renauld. *New Zealand J. Math.* 31, 73 (2002).

✱253 W. Reyes. *Nieuw Archief Wiskunde* 9, 299 (1991).

✱254 D. Richards. *Math. Mag.* 53, 101 (1980).

✱255 C. T. Ridgely. *Am. J. Phys.* 67, 414 (1999). *DOI-Link*

✱256 R. F. Rinehart. *Am. Math. Monthly* 62, 395 (1955).

✱257 L. Rodman in M. Hazewinkel (ed.). *Handbook of Algebra* v.1, Elsevier, Amsterdam, 1996. *BookLink*

✱258 A. Rogers, P. Loly. *Am. J. Phys.* 72, 786 (2004). *DOI-Link*

✱259 W. W. Rouse Ball, H. S. M. Coxeter. *Mathematical Recreations and Essays*, University of Toronto Press, Toronto, 1974. *BookLink (3)*

✱260 D. G. Saari. *The Geometry of Voting*, Springer-Verlag, New York, 1994. *BookLink (2)*

✱261 D. G. Saari. *Chaotic Elections! A Mathematicians Looks at Voting*, American Mathematical Society, Providence, 2001. *BookLink*

✱262 D. G. Saari. *Math. Mag.* 70, 83 (1997).

✱263 D. Sandell. *Math. Scientist* 16, 78 (1991).

✱264 L. San Martin, Y. Oono. *Phys. Rev.* E 57, 4795 (1998). *DOI-Link*

✶265 P. Schatte. *ZAMM* 53, 553 (1973).

✶266 A. Schenkel, J. Zhang, Y. C. Zhang. *Fractals* 1, 47 (1993).

✶267 E. Scholz. *arXiv:math.HO*/0409578 (2004). *Get Preprint*

✶268 C. Schmoeger. *Lin. Alg. Appl.* 359, 169 (2003). *DOI-Link*

✶269 E. Schmutzer. *Relativistische Physik*, Geest and Portig, Leipzig, 1968. *BookLink*

✶270 H. Schubert. *Zwölf Geduldspiele*, Göschen, Leipzig, 1899.

✶271 R. Sedgewick, P. Flajolet. *Analysis of Algorithms*, Addison Wesley, Reading, 1996. *BookLink*

✶272 R. Sharipov. *arXiv:math.DG*/0503332 (2005). *Get Preprint*

✶273 R. Shaw. *Int. J. Math. Edu. Sci. Technol.* 18, 803 (1987).

✶274 W. Sierpinski. *A Selection of Problems in the Theory of Numbers*, Pergamon, New York, 1964. *BookLink*

✶275 W. Sierpinski. *Elementary Theory of Numbers*, North Holland, Amsterdam, 1988. *BookLink (2)*

✶276 Z. K. Silagadze. *Complex Systems* 11, 465 (1997).

✶277 Z. K. Silagadze. *arXiv:physics*/9901035 (1999). *Get Preprint*

✶278 Z. K. Silagadze. *arXiv:hep-ph*/0106235 (2001). *Get Preprint*

✶279 B. Sing. *arXiv:math-ph*/0207037 (2002). *Get Preprint*

✶280 J. Skilling. *Phil. Trans. R. Soc. Lond.* 278, 15 (1975).

✶281 J. Skilling in J. Skilling (ed.). *Maximum Entropy and Bayesian Methods*, Kluwer, Dordrecht, 1989. *BookLink*

✶282 M. A. Snyder, J. H. Curry, A. M. Dougherty. *Phys. Rev.* E 64, 026222 (2001). *DOI-Link*

✶283 E. Stade. *Rocky Mountain J. Math.* 29, 691 (1999).

✶284 P. S. Stanimirović, M. B. Tasić. *Appl. Math. Comput.* 135, 443 (2003). *DOI-Link*

✶285 R. P. Stanley. *Enumerative Combinatorics*, Cambridge University Press, Cambridge 1999. *BookLink (5)*

✶286 H. M. Stark. *An Introduction to Number Theory*, Markham, Chicago, 1970. *BookLink*

✶287 E. Stensholt. *SIAM Rev.* 38, 96 (1996).

✶288 T. J. Stieltjes. *J. reine angew. Math.* 89, 343 (1880).

✶289 Y. Stolov, M. Idel, S. Solomon. *arXiv:cond-mat*/0008192 (2000). *Get Preprint*

✶290 F. J. Studnička. *Monatsh. Math.* 10, 338 (1899).

★291 Z.-W. Sun. *Discr. Math.* 257, 143 (2002). *DOI-Link*

★292 A. Taivalsaari. *ACM Comput. Surv.* 28, 438 (1996). *DOI-Link*

★293 S.-I. Takekuma. *Hitotsubashi J. Econom.* 38, 139 (1997).

★294 J.-I. Tamura in V. Berthé, S. Ferenczi, C. Mauduit, A. Siegel (eds.). *Substitutions in Dynamics, Arithmetics and Combinatorics*, Springer-Verlag, Berlin, 2002. *BookLink*

★295 V. Tapia. *arXiv:gr-qc*/0408007 (2004). *Get Preprint*

★296 A. D. Taylor. *Mathematics and Politics*, Springer-Verlag, New York, 1995. *BookLink (2)*

★297 A. D. Taylor. *Am. Math. Monthly* 109, 321 (2002).

★298 C. R. Tolle, J. L. Budzien, R. A. LaViolette. *Chaos* 10, 331 (2000). *DOI-Link*

★299 L. N. Trefethen, D. Bau, III. *Numerical Linear Algebra*, SIAM, 1997. *BookLink (2)*

★300 G. Troll, P. beim Graben. *Phys. Rev.* E 57, 1347 (1998). *DOI-Link*

★301 M. Trott. *The Mathematica GuideBook for Graphics*, Springer-Verlag, New York, 2004. *BookLink*

★302 M. Trott. *The Mathematica GuideBook for Numerics*, Springer-Verlag, New York, 2005. *BookLink*

★303 M. Trott. *The Mathematica GuideBook for Symbolics*, Springer-Verlag, New York, 2005. *BookLink*

★304 B. Tsaban. *arXiv:math.NA*/0204028 (2003). *Get Preprint*

★305 C. Tsallis. *arXiv:cond-mat*/9903356 (1999). *Get Preprint*

★306 C. Tsallis, M. P. de Albuquerque. *arXiv:cond-mat*/9903433 (1999). *Get Preprint*

★307 C. Tsallis. *Anais Acad. Brasil. Ciências* 74, 393 (2002).

★308 L. U. Uko. *Math. Scientist* 18, 67 (1993).

★309 C. Van den Broeck, J. M. R. Parrondo. *Phys. Rev. Lett.* 71, 2355 (1993). *DOI-Link*

★310 I. Vardi. *The Mathematica Journal* 1, n3, 53 (1991).

★311 R. Vein, P. Dale. *Determinants and Their Applications in Mathematical Physics*, Springer-Verlag, New York, 1999. *BookLink*

★312 G. Venkatasubbiah. *Math. Student* 7, 101 (1940).

★313 P. Vignolo, A. Minguzzi, M. P. Tosi. *Phys. Rev. Lett.* 85, 2850 (2000). *DOI-Link*

✱314 D. Wagner. *The Mathematica Journal* 6, n1, 54 (1996).

✱315 Y. H. Wang, L. Tang, Y. S. Lou. *Math. Scientist* 24, 96 (1999).

✱316 D. S. Watkins. *SIAM Rev.* 34, 427 (1982).

✱317 J. J. Wavrik. *Comput. Sc. J. Moldova* 4, 1 (1996).

✱318 S. Weinberg. *The Quantum Theory of Fields* v.1, Cambridge University Press, Cambridge, 1996. *BookLink*

✱319 A. Weinmann. *J. Lond. Math. Soc.* 35, 265 (1960).

✱320 T. West. *Comput. Phys. Commun.* 77, 286 (1993). *DOI-Link*

✱321 H. Weyl. *The Theory of Groups and Quantum Mechanics*, Dover, New York, 1931. *BookLink*

✱322 R. Wheeldon, S. Counsell. *arXiv:cs.SE*/0305037 (2003). *Get Preprint*

✱323 D. V. Widder. *Trans. Am. Math. Soc.* 30, 126 (1928).

✱324 J. H. Wilkinson. *The Algebraic Eigenvalue Problem*, Oxford, Clarendon, 1965. *BookLink (2)*

✱325 F. Wille. *Humor in der Mathematik*, Vandenhoeck & Ruprecht, Göttingen, 1987. *BookLink*

✱326 D. Withoff. *The Mathematica Journal* 4, n2, 56 (1994).

✱327 S. Wolfram. *Rev. Mod. Phys.* 55, 601 (1983). *DOI-Link*

✱328 H. Wolkowicz, G. P. H. Styan. *Lin. Alg.* 29, 471 (1980).

✱329 D. R. Woodall. *Math. Intell.* 8, n4, 36 (1986).

✱330 G. Xin. *arXiv:math.CO*/0409468 (2004). *Get Preprint*

✱331 S. Y. Yan. *Number Theory for Computing*, Springer-Verlag, Berlin, 2000. *BookLink (2)*

✱332 A. C.-C. Yang, C.-K. Peng, H.-W. Yien, A. L. Goldberger. *Physica* A 329, 473 (2003). *DOI-Link*

✱333 C. Yu, H. Song. *arXiv:math-ph*/0412060 (2004). *Get Preprint*

✱334 *ZEIT magazin* 4.9.1992 page 68 LOGELEI VON ZWEISTEIN (1992).

✱335 D. Zeitlin. *Am. Math. Monthly* 65, 345 (1958).

✱336 Y. Z. Zhang. *Special Relativity and Its Experimental Tests*, World Scientific, Singapore, 1997. *BookLink*

✱337 L. Zhipeng, C. Lin, W. Huajia. *arXiv:math.ST*/0408057 (2004). *Get Preprint*

✱338 G. K. Zipf. *Human Behavior and the Principle of Least Effort*, Addison-Wesley, Cambridge, 1949. *BookLink*