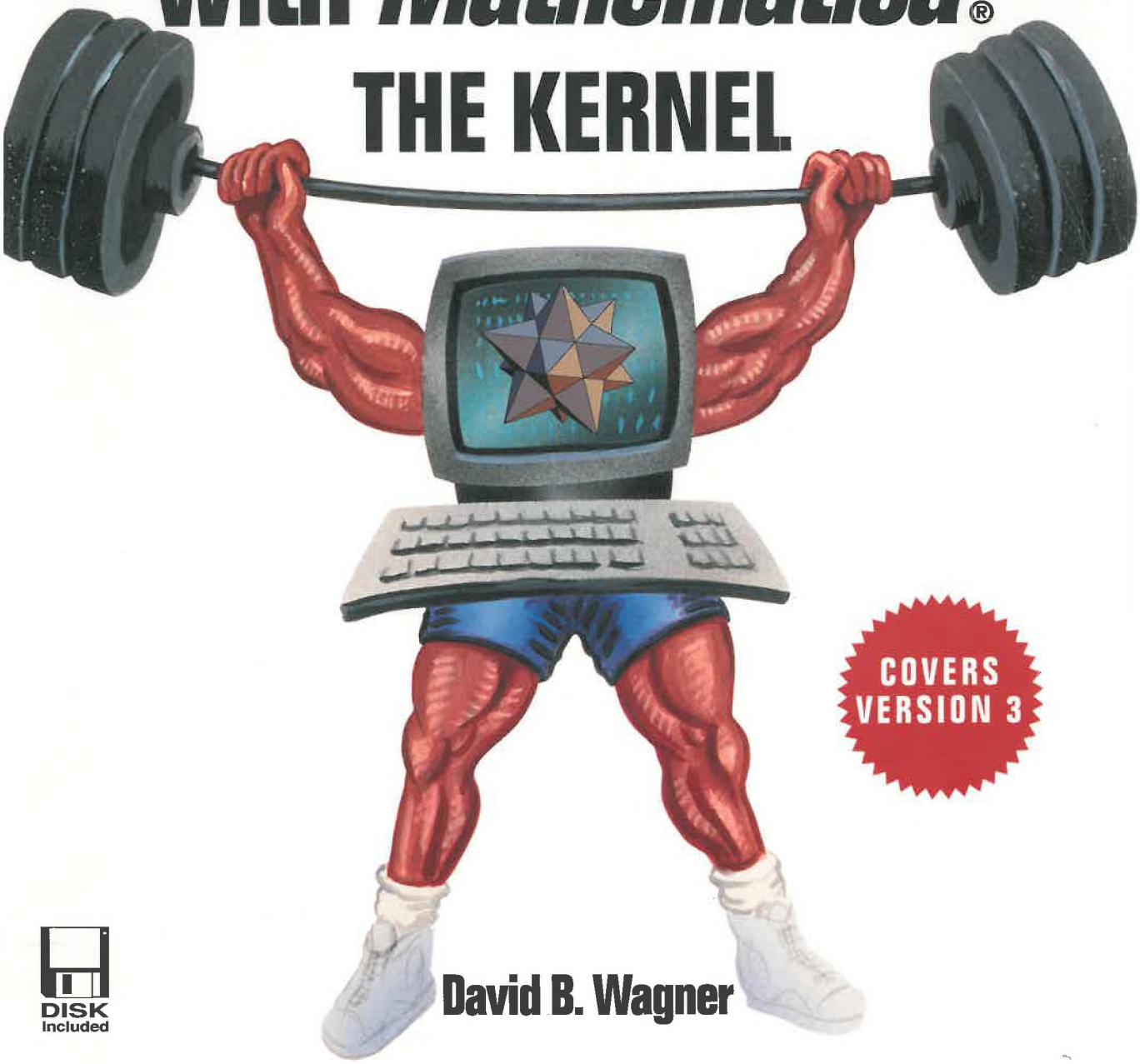
Computing McGraw-Hill

Power Programming with *Mathematica*®

THE KERNEL



COVERS VERSION 3

DISK Included

David B. Wagner

Power Programming with *Mathematica*

Disclaimer & Limitation Liability

McGraw-Hill makes no representations or warranties as to the accuracy of any information contained in the McGraw-Hill Material, including any warranties of merchantability or fitness for a particular purpose. In no event shall McGraw-Hill have any liability to any party for special, incidental, tort, or consequential damages arising out of or in connection with the McGraw-Hill Material, even if McGraw-Hill has been advised of the possibility of such damages.

Related Titles

Buehrens • DataCAD
Dorfman • C++ by Example
Giencke • Portable C++
Hatton • Safer C
Hood • Easy AutoCAD for Windows
Machover • CAD/CAM Handbook
Muller • Webmaster Guide to HTML
Shammas • C/C++ Mathematical Algorithms for Scientists and Engineers
Shammas • Mathematical Algorithms in Visual Basic for Scientists and Engineers
Tanner • Practical Queueing Theory
Zetie • Practical User Interface Design

To order or receive additional information on these or any other McGraw-Hill titles, in the United States please call 1-800-822-8158. In other countries, contact your local McGraw-Hill representative.

Key = WM16XXA

Power Programming with *Mathematica*

The Kernel

David B. Wagner
Principia Consulting
Boulder, Colorado

McGraw-Hill
New York San Francisco Washington, D.C. Auckland Bogotá
Caracas Lisbon London Madrid Mexico City Milan
Montreal New Delhi San Juan Singapore
Sydney Tokyo Toronto

Library of Congress Cataloging-in-Publication Data

Wagner, David B.
Power programming with Mathematica : the Kernel / David B. Wagner.
p. cm.
Includes bibliographical references and index.
ISBN 0-07-912237-X
1. Mathematica (Computer program language) I. Title.
QA76.73.M29W34 1996
510'.285'5133—dc20 96-18798
CIP

McGraw-Hill
A Division of The McGraw-Hill Companies

1 2 3 4 5 6 7 8 9 0 AGM/AGM 9 0 1 0 9 8 7 6

P/N 067679-8
PART OF
ISBN 0-07-912237-X

The sponsoring editor for this book was John Wyzalek, the editing supervisor was Stephen M. Smith, and the production supervisor was Pamela A. Pelton.

Printed and bound by Quebecor/Martinsburg.

McGraw-Hill books are available at special quantity discounts to use as premiums and sales promotions, or for use in corporate training programs. For more information, please write to the Director of Special Sales, McGraw-Hill, 11 West 19th Street, New York, NY 10011. Or contact your local bookstore.

This book is printed on acid-free paper.

Contents

Preface ix

Part 1: Preliminaries

1 Introduction 3

1.1 Programming in *Mathematica* 4
1.2 Power Programming Examples 7
1.3 Where to Go Next 15

2 Language Fundamentals 17

2.1 Expressions 17
2.2 Evaluation of Expressions 25
2.3 Special Input Forms 30

3 Lists and Strings 49

3.1 List Basics 49
3.2 Generating Lists 52
3.3 Listable Functions 54
3.4 Getting Information about Lists 56
3.5 Manipulating Lists 58
3.6 Character Strings 63
3.7 Appendix: Listable Functions 67

Part 2: Programming Techniques

4 Procedural Programming 71

4.1 Functions 71
4.2 Conditional Execution 80
4.3 Iteration 83
4.4 Parameter-Passing Semantics 88
4.5 Advanced Topic: Scoping 93

5 Functional Programming 97

5.1 Basic Functional Programming 98
5.2 Variations on a Theme 107
5.3 Iterating Functions 114
5.4 Recursion 127
5.5 Manipulating Normal Expressions 131
5.6 Additional Resources 138
5.7 Appendix: Lisp–*Mathematica* Dictionary 139

6 Rule-Based Programming 141

6.1 Patterns 141
6.2 Rules and Functions 147
6.3 Pattern Building Blocks 153
6.4 Dynamic Programming 165
6.5 Overriding Built-in Functions 175
6.6 Additional Resources 184

7 Expression Evaluation 185

7.1 The Evaluation Process 185
7.2 Nonstandard Evaluation 197
7.3 Working with Held Expressions 208
7.4 Additional Resources 225
7.5 Appendix: Functions with Hold- Attributes 225

Part 3: Extending the System

8 Writing Packages 229

8.1 Contexts 229
8.2 Package Mechanics 235
8.3 Stylistic Considerations 241
8.4 Advanced Topic: Shadowing 243
8.5 Additional Resources 256

9 Details, Details 257

9.1 Diagnostic Messages 257
9.2 Options 262
9.3 Numerical Evaluation 266
9.4 Custom Output Formats 274
9.5 Respect Existing Definitions 279
9.6 Application: Defining a New Data Type 283
9.7 Additional Resources 293

Part 4: Programming for Performance

10 Performance Tuning 297

10.1 Rules of Thumb 298
10.2 Procedural Perils 305
10.3 Recursion Risks and Rewards 314
10.4 Rewrite Rules 320
10.5 Compiled Functions 325
10.6 Additional Resources 334

11 *MathLink* 335

11.1 *MathLink* Fundamentals 336
11.2 Template-Based *MathLink* Programs 341
11.3 Debugging *MathLink* Programs 349
11.4 Manual Data Handling 356
11.5 Integrating Installable Functions and Packages 365
11.6 Callbacks to the Kernel 369
11.7 Error Checking 372
11.8 Making Installed Functions Abortable 377
11.9 Miscellaneous *MathLink* Data Types 378
11.10 Additional Resources 384

Part 5: Miscellanea

12 Input/Output 387

12.1 File and Directory Management 387
12.2 High-Level Output 388
12.3 Low-Level Output 392
12.4 High-Level Input 396
12.5 Low-Level Input 402
12.6 Additional Resources 404

13 Debugging 405

13.1 Tracing Evaluations 405
13.2 Interactive Debugging 413
13.3 Additional Resources 421

Bibliography 423

Index 427

Preface

About *Mathematica*

Mathematica has become phenomenally popular in the last few years for its sophisticated numeric and symbolic mathematical capabilities. A lesser-known feature of *Mathematica* is its very elegant programming language. This language allows virtually unlimited extension of the system's capabilities to solve problems in special areas of interest. In fact, hundreds of *Mathematica packages*, embodying applications ranging from airfoil design to *Z* transforms, are available from public Internet sites.

Mathematica offers a combination of features that is unmatched by traditional program development systems such as C, Fortran, and Lisp, including:

- An interpretive environment for fast prototyping
- Compilation for speed
- An API (applications programming interface) for communicating with external programs written in compiled languages
- Functional and rule-based programming styles
- Abstract data typing and modularity
- Typeset mathematical input and output (version 3.0)
- Seamless integration with the computational and graphical capabilities of *Mathematica* (a partial list: exact or arbitrary-precision arithmetic (real or complex); linear algebra; symbolic and/or numerical evaluation of derivatives, integrals, and differential equations; hundreds of built-in functions from number theory, combinatorics, probability, statistics, and physics; 2-D and 3-D line, contour, surface, and vector plots).

Why I Wrote This Book

There are scores of *Mathematica* books available today; why bring yet another one into the world? I wrote this book to give *Mathematica* users a comprehensive source for learning how to program in *Mathematica*. There are two parts of this statement that I want to stress: *comprehensive* and *learning how to program.*

Regarding learning how to program: The majority of existing books about *Mathematica* are example-based texts in a particular area of application. Some appear to be programming books, but typically the author covers just enough about *Mathematica* programming to get to "the good stuff" — that is, applying *Mathematica* to his or her

area of specialization. I, on the other hand, am a computer scientist, not a mathematician, physicist, or engineer. To me, the programming *is* the good stuff. This book is a reflection of how a computer scientist sees *Mathematica* — a viewpoint that is, in my opinion, sorely under-represented in the existing literature.

Regarding comprehensiveness: Certain topics, such as debugging, performance tuning, and *MathLink*, are almost always given short shrift in the existing literature — and there is no single source that discusses all of them.[1] This book devotes an entire chapter to *each* of those topics. Furthermore, many of the advanced *Mathematica* programming techniques found here have not appeared in any other book. I have spent the last two years scouring many sources for this information, including Wolfram Research technical reports, conference papers, journal articles, packages, and Internet discussion groups. I also have discussed many issues directly with *Mathematica*'s developers.

As a result, I believe that this book makes a needed contribution to the *Mathematica* literature. Almost any *Mathematica* user, at any level of expertise, should find things in this book that she does not already know. At the same time, the book does not skip over the fundamentals, so it is accessible to persons who are just getting their feet wet in *Mathematica* programming.

Why You Need This Book

The audience for this book consists of *Mathematica* users who want to start writing programs, or who simply have a nagging feeling that the ways they solve their problems could be improved. In addition, *Mathematica* programmers who want to write significant extensions to the system will find this book valuable.

Subtleties of the *Mathematica* programming language

There are three programming paradigms that *Mathematica* supports: *procedural*, which is Fortran- or C-like; *functional*, with which you might have some experience if you've every programmed in Lisp; and *rule-based*, which is typified by Prolog. Most scientists and engineers, who are *Mathematica*'s principal user base, have no experience with any but the first of these paradigms.

Because *Mathematica* provides procedural programming constructs, a programmer with a traditional background is likely to fall into the trap of programming the same way he always has. Unfortunately,

- Different ways of performing an equivalent computation can vary dramatically in speed.
- The best way of doing something is almost never the procedural way, and thus is not obvious to someone who is used to traditional programming languages.

1. Says one restaurant patron to another: "The food at this restaurant is terrible!" Replies his companion: "Yes, and the portions are so *small!*"

- Pitfalls exist, of which a programmer used to other languages will not be aware.

There is near-universal agreement among the *Mathematica cognoscenti* that procedural programming is "bad" and that the functional and rule-based paradigms are to be preferred. I tend to agree. However, I also am of the opinion that procedural programming in *Mathematica* is an important steppingstone to the less familiar techniques. Therefore, this book devotes a chapter to procedural programming very early on, rather than leaving it as an afterthought or omitting the topic entirely. In a further attempt to ease the transition as much as possible, analogies with other programming languages are drawn whenever appropriate.

The expanding universe of *Mathematica* literature

Another problem that the aspiring *Mathematica* programmer faces is information overload. There simply is too much information available about *Mathematica*. Even if you never stray from the documentation that comes with the program, you are faced with Stephen Wolfram's 900-page treatise (1400 pages in version 3.0!), the 450-page *Guide to Standard Packages*, and the *User's Guide* that describes the notebook interface.

Add 50+ books, three periodicals, an Internet newsgroup and *MathSource*, and it becomes overwhelming. You don't have time to keep up with all of it because *Mathematica* is just a tool that you use for your real job. But keeping up with the *Mathematica* information glut *is* my job. You will find many specific references to the rest of the *Mathematica* literature throughout this book — not simply an exhaustive list of all known books on the subject at the very end.

New features of *Mathematica*

This book is current as of version 3.0 of *Mathematica*. Techniques or examples that are specific to version 3.0 are highlighted by the icon appearing to the left of this paragraph. Nearly everything else applies equally to version 2.2 or version 3.0.

Version 3.0 contains so many new features that it would have been impossible to discuss them all in a book of this size. Therefore, my editors and I decided that there will be a follow-on volume to this book. The volume you are holding covers only the *Mathematica* kernel, whereas the next volume will cover the *Mathematica* user interface. There are sound reasons for splitting the material along these lines. Since the design of the kernel is so mature, most of what is contained in this book also applies to earlier versions of *Mathematica*. On the other hand, the new features of the *Mathematica* user interface are so extensive — typeset input and output, new notebook file formats, programmable user-interface elements, etc. — that the second volume will be version 3.0-specific. There is one topic that has been split across the two volumes, and that is *MathLink*, Wolfram Research's interprogram communication API. This is because *MathLink* has two fundamentally different uses: to allow the kernel to call

functions written in compiled languages such as C (discussed in this volume), and to provide an alternative interface to the kernel (to be discussed in the next volume).

How to Use This Book

Road map

This book is designed to be read in a linear fashion; it does *not* consist of a series of stand-alone examples like some other books. Nevertheless, many people are going to want to pick and choose from the available material. Section 1.3, "Where to Go Next," gives a broad overview of the book.

Before going any further

If you don't already own a copy of *The Mathematica Book* ([Wolfram 91] for versions prior to 3.0, [Wolfram 96] for version 3.0 and later), go out right now and buy one. You can't put it off forever, and it can save you untold amounts of frustration in certain circumstances. It is the *final authority* on *Mathematica*.

This book uses the generic term "*The Mathematica Book*" to refer to either [Wolfram 91] or [Wolfram 96], whichever is appropriate to the version of *Mathematica* that you are using. For those cases in which the location of a reference differs between the two books, a more specific citation of the form "[Wolfram 91] §A.B or [Wolfram 96] §X.Y" will be used instead.

Pedagogical notes

Here are a few notes to give you some idea of what to expect as you read this book. For the most part, the presentation style is based on what I have found to be pedagogically effective during my five years as a university professor.

- In contrast to some authors' preferences for presenting a completely developed piece of code and then explaining it, I prefer to integrate the development of the code into the presentation. This mirrors the way persons typically develop code in an interactive programming environment such as *Mathematica*. Presenting the development step by step, rather than as an *a posteriori* discussion, also makes it less likely that explanations of crucial steps are omitted.
- In those cases in which showing the final version of the code under development would be too repetitive, the "finished product" can be found on the supplemental diskette.
- During the development of an example, I often show the "dead ends" that other authors leave out. I believe that these dead ends, when properly explained, are valuable examples in and of themselves. Besides, except in the most trivial cases no one writes a perfect program on the very first try, and it's unrealistic to give the reader the impression that he should be able to do so.

- Many of the smaller examples are presented several times throughout the book, each time using the technique being discussed at that point. (This is especially true of the chapters in Part 2, "Programming Techniques.") Therefore, if you are an experienced *Mathematica* programmer, try not to be dismayed by the relative unsophistication of some of the earlier examples; delay your judgment of my programming abilities until you have seen the entire story!
- Nontrivial examples will be found interspersed with the rest of material, rather than being segregated into their own chapters. This is because these examples have been chosen to illustrate the techniques being discussed at that point. (The Huffman coding example in Chapter 5, which occupies close to 10 pages, is a good example of this.)
- There are copious exercises throughout the book. In most cases the exercises immediately follow the material to which they relate, rather than being grouped together at the end of a chapter. This makes the book suitable for use as a workbook. I recommend that you work through the exercises as you read the book, rather than separating the two activities.

Please note that this book does not teach you *how* to program; instead, it teaches you how programming in *Mathematica* is different from programming in other languages. It is conceivable that a programmer with no *Mathematica* experience whatsoever might want to use this book as a programming language guide. This should be possible, as I have made every attempt to make the book self-contained; but bear in mind that the scope of this book is restricted to *Mathematica*'s programming features, and so it omits most mathematics-specific topics (e.g., symbolic algebra and calculus) and graphics.

Typographic conventions

The names of user-interface elements, such as keys on the keyboard, menus, menu items, and dialog-box button names, appear in **boldface**. Directory and file names[2] appear in *italic* type.

Annotated *Mathematica* dialogues are interspersed throughout the body of the text as shown below.

Lucid, insightful comments appear here.

```
This is input to Mathematica
This is output from Mathematica
```

A nonindented body paragraph (like this one) that follows an example is logically part of the example or the paragraph that precedes the example.

Note that I do not follow the practice of showing *Mathematica*'s *In[]:=* and *Out[]=* annotations in front of the inputs and outputs. In my experience, the line numbers in the

2. Users of the DOS or Windows versions of *Mathematica* should realize that the filenames on their systems may be abbreviations of the names shown here.

user's session get out of sync with those in the text almost immediately, so they are not worth the space they take up on the page.

Within body text, *Mathematica* expressions always appear in the `input` typeface. User-supplied input to *Mathematica* expressions appear in the *`slanted input`* typeface. Pay special attention to the typeface of quotation marks: `"thing1"` means that the quotation marks are part of the expression, whereas "`thing2`" means that they are not. The latter form is used when there exists some possibility of confusion in distinguishing between the expression text and the surrounding body text. This typically is the case only when the expression text contains punctuation characters, such as "`x..`".[3] Square brackets are never added to function names unless the meaning being conveyed is that a function call is taking place (e.g., the `Random` function can be used to generate a uniformly distributed pseudorandom number like this: `Random[]`).

Points of special note will be indicated by the following icons:

- Common mistakes and especially dangerous traps are indicated by the "be alert" icon shown to the left of this paragraph.

- Particularly useful techniques and tricks are indicated by the "idea" icon shown to the left of this paragraph.

- Material that appears in the enclosed electronic supplement is indicated by the diskette icon shown to the left of this paragraph. Although not explicitly indicated, answers to most of the exercises appear in the supplement as well.

- Techniques and examples that are specific to version 2.2 or earlier of *Mathematica* are flagged by the icon shown to the left of this paragraph.

- Techniques and examples that are specific to version 3.0 or later of *Mathematica* are flagged by the icon shown to the left of this paragraph.

Electronic supplement

The electronic supplement contains four major components: answers to most of the exercises, source code for the *MathLink* programs in Chapter 11, data files for the examples in Chapter 12, and packages that are developed throughout the book.

Since there is a new notebook file format in version 3.0 that is not backward compatible with version 2.2, all of the notebook files on the diskette are in the version 2.2 format. You can open these notebooks using either version 2.2 or version 3.0 of the

3. Furthermore, the standard typographic convention of placing a sentence-ending period within quotation marks has to be broken in this particular example, or else the purpose of the quotation marks is defeated.

Mathematica front end. Furthermore, although the notebooks were created using the MacOS version of *Mathematica*, the diskette itself is in MS-DOS format, so that it can be used on the widest possible range of computer systems.[4]

Contacting the Author

I would greatly appreciate hearing your comments about any of the material in this book. This includes corrections, requests for clarification, and requests for additional topics that you would like to see in a future edition. Please address all correspondence to the author via the Internet at:

```
dbwagner@princon.com
```

And while you're on-line, don't forget to check out PRINCIPIA CONSULTING's World-Wide Web site at:

```
http://www.princon.com/princon
```

Corrections to the book and the electronic supplement, and answers to frequently-asked questions, will be posted there as they are discovered. I ask that you please check that site before contacting me with a bug report or a question.

Colophon

This book was produced on a Power Macintosh using versions 2.2.2 and 3.0β1 of *Mathematica*, and version 4.0.4 of FrameMaker. The *Mathematica* notebooks were translated into Maker Interchange Format files using a proprietary filter developed by me. The body text face is Times, the section head and side head face is Optima, the chapter number face is Avant Garde, and the face used for computer voice is a semicondensed version of Prestige Elite with a few modifications (made using the font-editing program FontMonger).

Acknowledgments

First of all, a disclaimer: This list may be incomplete due to failing memory on my part; I don't mean to slight anyone who helped me out!

I have a lot of WRI people to thank, but particularly John Fultz, Todd Gayley, Theo Gray, Rob Knapp, Roman Maeder, K.J. Paradise, Shawn Sheridan, Robby Villegas, Tom Wickham-Jones, and Dave Withoff. Without their willingness to discuss technical issues, often in painstaking detail, this book would have been a lot less authoritative.

4. MacOS users: If your computer won't recognize this diskette, install the *PC Exchange* control panel and restart your machine. If your version of the system doesn't have this control panel, you will need to obtain a program called *Apple File Exchange* from Apple Computer, Inc.

Howard Berg and Jamie Peterson kept me supplied with all the *Mathematica* software, books, CD-ROMs, and, most important, T-shirts[5] that I needed to complete this book. In general, I was amazed at how fast everyone at WRI responded to my questions and concerns.

Extra-special thanks are due to Jack Goldberg, who reviewed the entire manuscript and provided lots of valuable feedback. Troels Petersen also helped me to improve my writing style.

I also want to acknowledge the (unwitting) help of the entire cast of characters of the `comp.soft-sys.math.mathematica` newsgroup, for constantly coming up with challenging problems for me to work on, many of which became examples in this book. They are far too numerous to list individually, but special recognition is owed to Ronald Bruck, who posed the question of how to establish a default thickness for lines in plots (Section 6.5.3, "Application: A default thickness for Plot"), and James Larson, who raised the idea of cross-referencing the symbols in a *Mathematica* program (Section 7.3.7, "Application: Dependency analysis").

Finally, I want to thank my editors at McGraw-Hill: Marjorie Spencer, who was willing to take a chance on an unknown author; and John Wyzalek, who picked up the ball midway through the project and kept things rolling. Both are great people to work with!

David B. Wagner
Boulder, Colorado
June 1996

5. When you look good, you feel good!

Power Programming with *Mathematica*

Part 1
Preliminaries

1

Introduction

Different people have different ideas of what it means to program in *Mathematica*. To some, programming in *Mathematica* means using the myriad of powerful built-in functions to create "one-liners" that perform nontrivial calculations with a minimum of effort. To others, it means defining new functions that extend the capabilities of the system in some way. More advanced users want to place their function definitions in separate files, called *packages*, that encapsulate the details of the implementation.

This book is intended to be a comprehensive reference to programming in *Mathematica*. In addition to all of the above, we attempt to assemble under a single cover many advanced techniques that either are scattered around the existing literature or can't be found at all. These techniques, which we group under the moniker *power programming*, deal with issues such as: overriding built-in functions; making user-defined functions behave as much like the built-in ones as possible; adding new abstract data types; changing the global behavior of the system; tuning *Mathematica* code for performance; and using the *MathLink* interprogram communication protocol. We also devote an entire chapter to the much-neglected topic of debugging *Mathematica* programs.

This introduction takes you on a tour through ever increasingly advanced programming examples, beginning with the simplest imaginable and concluding with a demonstration of some of the more sophisticated programs developed in this book. In the course of doing so, the author hopes to provide the reader with a better idea of what *power programming* is all about, and how this book differs from existing ones.

If your experience with *Mathematica* is very limited, you may wish to review the basic syntax of the language in the next chapter before continuing with this one.

1.1 Programming in *Mathematica*

Mathematica is an interactive system for doing mathematical calculations by computer. *Mathematica* can also be considered a programming language. Because *Mathematica* is interactive, however, it's often unclear what constitutes "programming" and what does not. For example, is

```
2 + 3
```
```
5
```

a program? Technically speaking, yes, but most users would not consider it so. How about:

```
a = 2;
b = 3;
a + b
```
```
5
```

There are three separate "statements" here; assignments are made to "variables" and then those "variables" are used in further calculations. Nevertheless, most *Mathematica* users probably would be too modest to consider this to be a program either. Then how about this:

```
f[x_] := x + 3
f[2]
```
```
5
```

Surely, this is a program: A function of a single argument is declared and then called.

As the *Mathematica* user gains expertise, her programs become more involved and, hopefully, more sophisticated as well. For example, the naïve *Mathematica* programmer might write the following code to compute the moving average of a list of numbers:

The *k*th moving average of a list of numbers averages every group of $k+1$ contiguous numbers in the list.

```
MovingAverage1[z_, k_] :=
Module[{zav = {}, i, j, temp},
    Do[ temp = 0;
        Do[temp += z[[i + j]], {j, 0, k}];
        AppendTo[zav, temp/(k + 1)],
        {i, 1, Length[z] - k}
    ];
    zav
]

nums = {1, 2, 4, 8, 16};
MovingAverage1[nums, 1]
```
```
{3/2, 3, 6, 12}
```

This is an example of a procedural style of programming; it is analogous to how one might perform this computation in a language such as C or Fortran. A more experienced *Mathematica* programmer might accomplish the same thing this way:

```
MovingAverage2[z_, k_] :=
    Plus @@ Partition[z, Length[z] - k, 1]/(k + 1)
```

This is an example of a functional style of programming (Chapter 5), which may be familiar to Lisp programmers. The functional program is more elegant and more efficient as well (see Section 1.2.6, "Program performance," for a performance comparison of these two functions). Furthermore, the author had to tweak the procedural version to get a bug or two out of it; the functional version worked the first time (this is true!). While this is of course merely anecdotal evidence for the superiority of the functional approach, it is consistent with the author's experience.

As another example, here is a procedural program to effect a transformation on the y values of a list of (x, y) data points:

But a more "*Mathematica*-like" way to do this would be to use pattern matching and rule substitution (Chapter 6):

To many *Mathematica* users, what has just been demonstrated is, in a nutshell, what programming in *Mathematica* is all about. These techniques don't need to be applied to large, complex programs to be useful. On the contrary, their real value is measured in terms of how much more productive they enable one to be during almost any interactive *Mathematica* session; the skilled programmer uses one-liners like those just demonstrated to avoid a lot of wasted time and effort. This book, like others on the topic, covers these techniques in some detail (specifically in Part 2, "Programming Techniques").

But this is a very limited notion of what it means to program in *Mathematica*. It is analogous to programming in C without ever learning about the run-time library, separately compiled modules, input/output, command-line argument processing, and so on. Or, to draw another analogy, it is like writing nothing but text-based programs for a system with a graphical user interface library.

As extensive as it is, *Mathematica* can't be all things to all people "right out of the box." The designers of *Mathematica* addressed this problem by providing building

blocks for constructing *packages*, which are separate files containing (hopefully) reusable function definitions. A package can be loaded into a *Mathematica* session at any point, and the functions defined therein become available, just as though they were built into the system.

For example, there is a `MovingAverage` function defined in one of the *standard packages* that come with *Mathematica* [WRI 93b]. To use it, one simply loads the package as shown below:

```
Needs["Statistics`MovingAverage`"]
MovingAverage[nums, 1]
```

$$\left\{\frac{3}{2},\ 3,\ 6,\ 12\right\}$$

There's even a usage message, just as there should be for a built-in function.

```
?MovingAverage
MovingAverage[list,n] returns a list of the n-th
   moving averages of list.
```

It's easy to imagine how to write such a package — simply place the definition of the function in a separate file. However, there are certain conventions that packages should follow; these conventions are discussed in Chapter 8.

But the functionality of many of the standard packages seems to involve a leap beyond the capabilities of most *Mathematica* users, even those who fancy themselves "programmers." As a simple example, consider the standard package that provides statistical hypothesis-testing functions.

This tests the hypothesis that the mean of `nums` is equal to 2.

```
Needs["Statistics`HypothesisTests`"]
MeanTest[nums, 2]
OneSidedPValue -> 0.147779
```

`MeanTest` can take certain options, just like many built-in functions.

```
MeanTest[nums, 2, KnownVariance->11/10]
OneSidedPValue -> 0.121454
```

Default options can be changed, too.

```
SetOptions[MeanTest, KnownVariance->11/10];
MeanTest[nums, 2]
OneSidedPValue -> 0.121454
```

A diagnostic is issued when an error is encountered.

```
MeanTest[{1, 1, 1, 1}]
MeanTest::badargs:
   Incorrect number or type of arguments.
MeanTest[{1, 1, 1, 1}]
```

The last example is particularly interesting. Note that `MeanTest` somehow managed to issue a diagnostic message without actually evaluating!

The types of behaviors just illustrated — usage messages, optional parameters, default options, diagnostic messages, and so on — are what make a function look

"built-in." The techniques for creating behaviors like these will be discussed in Chapter 9, "Details, Details."

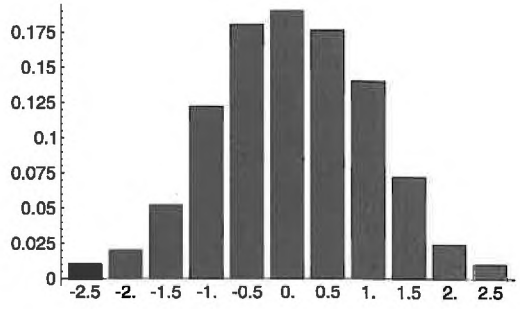
Here is one last example of the kinds of qualitatively harder things that the standard packages often do: overriding built-in functions. The built-in function `Random`, for example, is limited to generating uniformly distributed pseudorandom numbers.[1] However, a side effect of loading the *HypothesisTests.m* package is the loading of the *NormalDistribution.m* package, which defines the normal, chi-square, and Student's *t* distributions. The latter package also overrides the built-in `Random` function so that it can generate random variates having any of those distributions:

This generates 500 standard normal random variates.

```
norms = Table[Random[NormalDistribution[0, 1]], {500}];
```

To create a histogram, we need to load a couple more standard packages.

```
Needs["Statistics`DataManipulation`"]
Needs["Graphics`Graphics`"]
BarChart[BinCounts[norms, {-2.75, 2.75, .5}]/500.,
         BarLabels->Table[i, {i,-2.5, 2.5, .5}],
         DefaultFont->{"Helvetica", 7}];
```



Techniques for overriding built-in functions will be discussed in Chapter 6 and Chapter 9; the former is concerned solely with the mechanics of doing so, whereas the latter is concerned more with stylistic considerations.

1.2 Power Programming Examples

There is nothing magical about the standard packages; you don't have to work at Wolfram Research or know any secret handshakes to write programs like them. As proof of this, we will now demonstrate some of the more sophisticated programs that will be developed later in this book. If you're the kind of person who hates "spoilers," you might want to skip this section. On the other hand, if you are the kind of person who can't resist sneaking a peek at the last page of a mystery novel, or if you simply want a better idea of where we're heading, you'll want to read this first.

1. From this point on, the author will take the liberty of dropping the prefix "pseudo" from the term "pseudorandom."

Before proceeding, however, a warning: You are strongly urged to resist the temptation of skipping ahead to see how any of the programs here is implemented. If you do so, you will miss all of the material leading up to the development of that program. Thus, even though you may think you understand how a particular program works, at most you will have learned how to implement *that particular program;* whereas by reading the book linearly you will have learned the fundamental concepts that are transferable to programs of your own design.[2] To paraphrase an old proverb: The intent of this book is not to give you a fish (or two or three), rather, the intent is to teach you *how* to fish.

1.2.1 Optimal dot-product evaluation

The number of scalar multiplications required to evaluate the dot product of a chain of rectangular matrices can vary dramatically, depending on the order in which the subproducts are evaluated. The built-in `Dot` function does not take this into account. For example,

Here are three matrices; we wish to compute b1.b2.b3.

```
;
;
;
```

The default multiplication order takes this long ...

but multiplying in the optimal order (for this chain) takes only 1/20th the time!

In Section 6.4, "Dynamic Programming," we will write a function that calculates the optimal matrix-chain multiplication order. Both the type of algorithm (*dynamic programming*) and the technique used to implement it (recursion with *result caching*) are interesting in their own right. Later in that chapter (Section 6.5, "Overriding Built-in Functions") we will learn how to override the built-in `Dot` function so that it uses our function whenever the arguments to `Dot` consist of three or more matrices. Finally, after learning about packages in Section 8.2, "Package Mechanics," we will be able to encapsulate this code inside of a package:

```
Needs["OptimalDot`"]
```

Now `Dot` uses the optimal order.

```
Dot[b1, b2, b3]; // Timing
{0.433333 Second, Null}
```

2. And in anticipation of your next question: No, not all of the concepts embodied in a program are necessarily contained in a single chapter. Some programs draw on concepts learned in earlier chapters as well.

The extra execution time relative to the explicit specification of the optimal order is needed to figure out what the optimal order is. Since this may be a waste of time (e.g., if all of the matrices in the chain are square), the package adds an option called `Optimize` to `Dot`. The setting `Optimize->False` can be used to prevent the search for the optimal order.

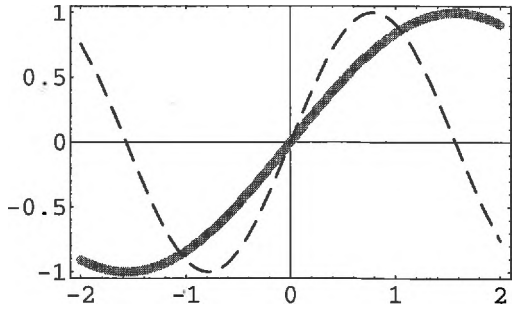
1.2.2 A default thickness for Plot

Another example of overriding a built-in function is the modification of the `Plot` function to give curves a default thickness when none is specified explicitly (Section 6.5.3, "Application: A default thickness for Plot"). Here is an example:

```
Needs["DefaultThickness`"]
```

The default thickness is *added* to the style of the gray curve, but the manifest thickness directive for the dashed curve is not overridden.

```
$DefaultThickness = .02;
Plot[{Sin[x], Sin[2x]}, {x, -2, 2},
    PlotStyle->{{GrayLevel[.7]},
                {Thickness[.005], Dashing[{.05, .03}]}},
    Frame->True
];
```



Naturally, the program also respects a `Thickness` directive specified by a default `PlotStyle` option, if one exists.

1.2.3 Finding symbol dependencies

In a very large program it can be difficult to anticipate the consequences of changing a small part of the code on the rest of the program. Many conventional languages have tools — such as call-graph analyzers and class browsers — that help the programmer understand the "big picture" structure of a program. In Section 7.3.7, "Application: Dependency analysis," we will develop a function called `dependson` that provides a first step in this direction for *Mathematica* programs.

The purpose of `dependson` is to take a symbol as an argument and return a list of all *nonlocal* symbols upon which that symbol depends. For example, given the following function definition,

```
f[x_] := Function[y, (y + z1) / (x + z2)]
```

the symbol `f` depends on the symbols `z1` and `z2` (as well as various arithmetic operators):

```
dependson[f]
{Plus, Power, Times, z1, z2}
```

Note that the symbol `x` is *local* to `f` and the symbol `y` is local to the `Function`, so those symbols do not appear in the list returned by `dependson[f]`. The `dependson` function is smart enough to recognize the difference between symbols appearing inside and outside of constructs such as `Function`. In the next example, the symbol `y` appearing outside of the `Function` refers to a nonlocal symbol `y`, *not* the `y` that appears as a formal parameter inside of the `Function`.

```
g[x_] := y + Function[y, x + y]
dependson[g]
{Plus, y}
```

Effecting behavior such as this relies on knowledge of the structure of expressions and techniques for manipulating expressions without allowing them to evaluate.

The `dependson` function developed in Section 7.3.7 works on only a limited class of expressions, and furthermore it finds only direct, not indirect, dependencies. (For example, if `f` depends on `g` and `g` depends on `h`, then `f` depends, indirectly, on `h`.) It is left as exercises for the reader to add features to `dependson`. The solutions to these exercises, and many more features, are integrated into a package called *Dependency-Analysis.ma* that is included on the supplementary diskette.

1.2.4 A package that prevents shadowing

Every *Mathematica* user has, at one time or another, made a mistake like the following:

You inadvertently refer to a package symbol before loading the package.

```
Show[Polyhedron[Dodecahedron]]
Show::gtype: Polyhedron is not a type of graphics.
Show[Polyhedron[Dodecahedron]]
```

You then load the package. Strange error messages appear.

```
Needs["Graphics`Polyhedra`"]
Polyhedron::shdw:
   Warning: Symbol Polyhedron
     appears in multiple contexts
    {Graphics`Polyhedra`, Global`}; definitions in
```

```
      context Graphics`Polyhedra`
      may shadow or be shadowed by other definitions.
Dodecahedron::shdw:
   Warning: Symbol Dodecahedron
     appears in multiple contexts
    {Graphics`Polyhedra`, Global`}; definitions in
     context Graphics`Polyhedra`
     may shadow or be shadowed by other definitions.
```

It appears that *Mathematica* still doesn't know about these symbols.

```
Show[Polyhedron[Dodecahedron]]
Show::gtype: Polyhedron is not a type of graphics.
Show[Polyhedron[Dodecahedron]]
```

This problem, which is called *shadowing*, results from the fact that the initial use of the symbol `Polyhedron` (and `Dodecahedron`) created a *different symbol with the same name* — i.e., there are now two symbols named `Polyhedron` — and the one being referred to by the name `Polyhedron` is the wrong one.[3]

In Section 8.4, "Advanced Topic: Shadowing," we will create a package called *AntiShadow.m* that "watches" for shadowing *while other packages are loading*. *AntiShadow.m* will automatically remove shadowing symbols from the `Global`` context, but only if those symbols have no definitions of any kind. To illustrate this, suppose that we had loaded *AntiShadow.m* before executing any of the above.[4]

```
Needs["AntiShadow`"]
```

Furthermore, suppose that one of the symbols used above actually had been given a value before we loaded the `Graphics`Polyhedra`` package.

```
Polyhedron = "a 3-D analogue of a polygon";
```

When the package is loaded, `Polyhedron` cannot be removed because it has a value.

```
Needs["Graphics`Polyhedra`"]
Polyhedron::shdw:
   Warning: Symbol Polyhedron
     appears in multiple contexts
    {Graphics`Polyhedra`, Global`}; definitions in
     context Graphics`Polyhedra`
     may shadow or be shadowed by other definitions.
```

On the other hand, `Dodecahedron` has no values, so it is removed automatically.

```
Dodecahedron::noshdw:
   Warning: the symbol Global`Dodecahedron
     has been removed from the global context to
     prevent shadowing.
```

3. Although this is probably very confusing to you at this point, you will learn about the mechanism that causes this behavior in Chapter 8.
4. If you are following along in your own *Mathematica* session as you read this, terminate the kernel and restart it before proceeding.

There are still two definitions of `Polyhedron`...

```
?*`Polyhedron
Polyhedron
Graphics`Polyhedra`Polyhedron
```

...but only one definition of `Dodecahedron` — the *right* one!

```
?*`Dodecahedron
Dodecahedron is a kind of polyhedron, for use with
   the Polyhedron function.
```

1.2.5 A new data type: Prime factorizations

In Section 9.6, "Application: Defining a New Data Type," we will create a package that defines a new data type called a `Factorization`.

```
Needs["Factorization`"]
```

```
?Factorization
Factorization[{n1, p1}, {n2, p2}, ...] represents the
   number n1^p1 * n2^p2 * ...
```

`Factorization` has a special print format.

```
Factorization[{3, 2}]
```

$$3^2$$

The package overrides the built-in `FactorInteger` function so that it returns a `Factorization`.

```
FactorInteger[9]
```

$$3^2$$

The package overrides all of the arithmetic operators (and quite a few other numerical functions) so that they "do the right thing" when given one or more `Factorizations` as arguments. Here are some examples:

```
% + 4^2
```

$$5^2$$

```
% + 1325
```

$$2\ 3^3\ 5^2$$

```
% ^ 2
```

$$2^2\ 3^6\ 5^4$$

```
1/%
```

$$2^{-2}\ 3^{-6}\ 5^{-4}$$

```
Sqrt[%]
```

$$2^{-1}\ 3^{-3}\ 5^{-2}$$

The package-defined function `ExpandFactorization` turns a `Factorization` back into a number.

```
ExpandFactorization[%]
 1
----
1350
```

Sometimes an operation turns a `Factorization` into an irrational number. Nevertheless, the number remains in factored form until converted by `ExpandFactorization`.

```
irr = Sqrt[FactorInteger[-3]]
-1^1/2 3^1/2
```

`ExpandFactorization` maintains the number in an exact form.

```
ExpandFactorization[irr]
I Sqrt[3]
```

The salient features of this example are the special output formatting and the large number of overridden functions. Special care has been taken to ensure that all of the new definitions work together smoothly, and that the new data type looks as much as possible as though it is built-in. For example, here is a detail that would have been easy to overlook:

A `Factorization` can be evaluated numerically with the `N` operator.

```
N[irr]
1.73205 I
```

1.2.6 Program performance

Many programmers assail *Mathematica* for its inefficiency relative to compiled languages such as C and Fortran. To be sure, there is a price to be paid for the power and flexibility of *Mathematica*'s interpreted execution and symbolic manipulation capabilities. What the critics do not realize, however, is that much of the perceived inefficiency of *Mathematica* is due to the use of ineffectual programming idioms from other languages; alternative techniques exist that are a better match for *Mathematica*.

As an example, consider the two moving average functions that were given as examples of programming style in Section 1.1. The first function, which has a decidedly procedural flavor to it (i.e., it smacks of C and Fortran), is terribly inefficient:

It takes over 30 seconds for `MovingAverage1` to compute 900 different 100-number averages!

```
nums = Range[1., 1000., 1.];
Timing[ MovingAverage1[nums, 100]; ]
{34.6333 Second, Null}
```

There are many reasons for the inefficiency of `MovingAverage1`; in fact, that function is something of a "straw man" in that it represents just about the poorest possible way of coding this algorithm in *Mathematica*. The fact that this happens to coincide with the way that is most likely to come first to the mind of the C or Fortran programmer is unfortunate; the author is convinced that this unhappy coincidence is the source

of a great deal of ill will toward programming in *Mathematica*. Instead of cursing the darkness, however, a much more productive thing to do is to shed some light on the subject, which we'll do in Chapter 10, "Performance Tuning." Armed with that knowledge, any *Mathematica* programmer can write functions that are orders of magnitude more efficient than their procedural counterparts. `MovingAverage2` provides an excellent example of this:

```
Timing[ MovingAverage2[nums, 100]; ]
{2.56667 Second, Null}
```

Another way to speed up a *Mathematica* function is to use the built-in compiler to translate the function into a special pseudo-code representation that can be executed very efficiently. Below we show a compiled version of `MovingAverage1`. The code contains several subtleties that will be explained in Section 10.5, "Compiled Functions."

```
MovingAverage3 = Compile[{{z, _Real, 1}, {k, _Integer}},
    Module[{zav = {0.}, i = 0, j = 0, temp},
        Do[ temp = 0.;
            Do[temp += z[[i + j]], {j, 0, k}];
            AppendTo[zav, temp/(k + 1)],
            {i, 1, Length[z] - k}
        ];
        Drop[zav, 1]
    ]
]

Timing[ MovingAverage3[nums, 100]; ]
{4.8 Second, Null}
```

While not quite as fast as `MovingAverage2`, `MovingAverage3` is nevertheless an order of magnitude faster than `MovingAverage1`, without requiring any substantive changes in programming style.

But even the most efficient *Mathematica* function is not going to be as fast as an equivalent function written in a compiled language such as C or Fortran. For the ultimate in speed, the *MathLink* protocol for communicating with externally compiled functions can be used. Here is an example of using *MathLink* to communicate with a moving average function very similar to `MovingAverage1`, except that this particular function has been coded in C and compiled externally:

The `Install` command launches the *MathLink* program.

```
link = Install["Moving Average"];
```

The externally compiled function can now be used as though it were built in.

```
Timing[ MLMovingAverage[nums, 100]; ]
{0.133333 Second, Null}
```

The result of the `Timing` command is misleading; it is the amount of time used by the *Mathematica* kernel, not the time used by the external function. The total elapsed time of the above operation was about 0.9 seconds. In this data-intensive example, the majority of the time is spent shuttling data back and forth between the kernel and the *MathLink* program. Functions with a higher ratio of computation to communication will of course enjoy an even larger relative improvement from external compilation.

MathLink also can be used to call the kernel from other programs — in fact, this is how the *Mathematica* front end communicates with the kernel! A discussion of *MathLink* is the topic of Chapter 11.

Uninstall causes the *MathLink* program to terminate.

```
Uninstall[link];
```

1.3 Where to Go Next

The remaining chapters in Part 1 cover the basic syntax and use of *Mathematica*. If you have a nontrivial amount of experience using *Mathematica*, you may find that you can skip this material. Be forewarned, however, that there are important concepts presented here that won't be mentioned explicitly again. You should probably at least skim the remainder of Part 1, and keep a lookout for the icons in the margin (described in the Preface). This will provide a reality check for you, to see if your self-assessment of your abilities is accurate. If you have used *Mathematica* to do numerical mathematics but not any programming, then you should still be sure to read Chapter 3, "Lists and Strings."

Part 2, "Programming Techniques," introduces the three fundamental programming paradigms supported by *Mathematica:* procedural (Chapter 4), functional (Chapter 5), and rule-based (Chapter 6). Needless to say, these chapters are a prerequisite for the remainder of the book and should be read, in the order presented, by all but the most expert *Mathematica* programmers. Unless you really are an expert, you will find the rest of the book to be *very heavy sledding* if you skip any of these chapters! Chapter 7, "Expression Evaluation," presents the author's version of a *Grand Unified Theory* of *Mathematica* — that is, an attempt to explain all of the diverse behaviors observed in earlier chapters in terms of a minimal collection of concepts. This chapter can be skipped on a first reading.

Part 3, "Extending the System," makes the quantum jump from writing small, stand-alone functions to writing integrated collections of functions and grouping them into packages. It also covers many techniques for making user-defined functions behave as the built-in ones do.

Part 4, "Programming for Performance," and Part 5, "Miscellanea," cover topics such as improving the performance of *Mathematica* code, the *MathLink* interprocess communication protocol, input/output, and debugging. Readers with a particular interest in any of those topics may skip directly there after completing Chapter 6.

2

Language Fundamentals

Mathematica is a *term-rewriting system.* Given an *expression* — which we will define in a precise manner below — as input, the fundamental operation performed by the *Mathematica* kernel is to recognize terms within the expression that it knows how to replace with other, hopefully simpler terms. For example, in the expression `a*a + D[a^3, a]`, *Mathematica* rewrites `a*a` as `a^2` (`^` is the exponentiation operator); then it rewrites `D[a^3, a]` as `3 a^2` (`D` is the derivative operator); and it then recognizes that `a^2 + 3 a^2` can be rewritten as `4 a^2`. It thus mimics the way that a human being does mathematics, although it does so in a completely algorithmic manner.

It turns out that expressions are the only type of object in *Mathematica* — they are used to represent both code and data. Furthermore, expressions have a recursive structure: Bigger expressions are composed of smaller expressions, which are in turn constructed of even smaller expressions, continuing in this manner until the subexpressions can be broken down no further — the *atoms* of the language. When *Mathematica* performs term rewriting, it always replaces one expression by another expression. This consistency of representation and operation is the key to the remarkable power of the *Mathematica* programming language.

2.1 Expressions

2.1.1 Normal expressions

Everything in *Mathematica* is an *expression.* There are fundamentally two types of expressions: *normal* expressions, which are of the form

```
head[part1, part2, ...]
```

where each of `head`, `part1`, `part2`, etc. is another expression; and *atoms*, which can be symbols, numbers, or character strings. (Atoms will be discussed in the next sec-

tion.) Thus, `Sin[Log[2.5, 7]]` is a normal expression having a single part; the head is an atom (the symbol `Sin`), and the single part is another normal expression, namely, `Log[2.5, 7]`. The latter expression has another symbol as its head (`Log`) and two parts: the real number `2.5` (which is interpreted as the base of the logarithm) and the integer `7`.

The syntax of expressions is designed to resemble the *function call* construct in languages such as C and Fortran. Associating symbolic heads such as `Sin` and `Log` with functions seems like a natural thing to do, and we will often refer to the head of an expression as a function and the parts of an expression as *arguments* of the function call. But not every normal expression can be thought of as a function call — an expression can also represent data. For example, `RGBColor[1, 0, 0]` is a *graphics directive* that tells *Mathematica* that the graphics *primitives* with which it is associated (which are themselves expressions) should be rendered in red. There is no "function" associated with the symbol `RGBColor`, and the expression `RGBColor[1, 0, 0]` cannot be rewritten in any way.

We assert here that every *Mathematica* expression can be constructed using only three syntactic building blocks: atoms, square brackets, and commas. If you have used *Mathematica* before, this may seem hard to believe because of the variety of syntax in the language. As a simple example, what about the expression given in the introduction to this chapter:

```
a*a + D[a^3, a]
```

$$4\ a^2$$

In fact, this expression could also have been entered as

```
Plus[Times[a, a], D[Power[a, 3], a]]
```

$$4\ a^2$$

Mathematica's parser converts input such as `a*a` into `Times[a, a]`, `a^3` into `Power[a, 3]`, and so forth. Syntactic forms such as `*`, `^`, and `+` are called *special input forms*. Without such forms, entering even moderately complex expressions would be unbearably tedious. We'll describe some of the most common special input forms in Section 2.3; additional forms will be introduced as needed throughout the book.

Note that not only do expressions with head `Plus`, `Times`, `Power`, etc. have special input forms, they also have special *output* forms. That is why the result of evaluating the above expression was printed in standard mathematical notation. You can force *Mathematica* to print the internal form of an expression by wrapping the head `FullForm` around it:

```
FullForm[4 a^2]
Times[4, Power[a, 2]]
```

2.1.2 Atoms

An *atom* is a *Mathematica* expression that cannot be broken down into smaller *Mathematica* expressions. There are three broad classes of atoms: *symbols*, *numbers*, and *character strings*.

Symbols

A *symbol* is a sequence of letters, digits, and the character $ that does not begin with a digit. Examples of symbols are `a`, `abc`, `a2`, `a2b`, `$a`, and `a$`. Though it may be tempting to think of symbols as being similar to *variables* in programming languages such as C or Fortran, you should try not to do so. Symbols are much more powerful because *symbols do not need to have any value assigned to them in order to be used in computations*. That is, a symbol can stand for itself; it is not merely a proxy for data.

Here is an example of a symbolic computation. The result is mathematically true for arbitrary values.

```
a + b - 2a
-a + b
```

All system-defined symbols begin with capital letters or the $ character (e.g., `Plus`, `D`, `FullForm`, `$Version`), so it's a good idea to begin user-defined symbols with lowercase letters to avoid confusion.

Numbers

There are four types of numbers in *Mathematica:* integers, which consist of a sequence of decimal digits *dddddd*; real numbers, which are of the form *ddd.ddd*; rational numbers of the form *integer1/integer2*; and complex numbers of the form *a* + *b* `I`, where *a* and *b* can be any of the other three types.

This is an integer.

```
1234567890
1234567890
```

This is a real number.

```
12345.67890
12345.6789
```

This is a rational number.

```
2/3
```

$$\frac{2}{3}$$

This is a complex number.

```
2/3 + 4.5I
```

$$\frac{2}{3} + 4.5\ \mathrm{I}$$

Any of the numeric types can have a virtually unlimited number of digits.

This is 5 raised to the power 73.

```
5^73
1058791184067875423835403125849552452564239501953125
```

The above calculation is an example of *exact* arithmetic: *Mathematica* assumes that the input 5 means "the exact integer 5" and it uses as many digits as are necessary to get an exact answer. Compare that with the following:

```
5.000000000000000000000000^73
```

$$1.058791184067875423835 4 \times 10^{51}$$

The presence of a decimal point in the input is taken to mean that the input is approximate and is known only to as many digits as have been explicitly entered. Therefore, *Mathematica* performs the computation using *arbitrary-precision* arithmetic, keeping track of how many digits in the answer are justified by the number of significant figures in the input and the operations that are being performed. In the present example, the two least-significant digits have been lost:

The `Precision` function returns the number of significant digits in an approximate number.[1]

```
Precision[5.000000000000000000000000]
25

Precision[5.000000000000000000000000^73]
23
```

The precision of an exact number is, by definition, infinite.

```
Precision[5]
Infinity
```

In version 3.0 you can specify the precision of an approximate number explicitly by using the syntax *`number`precision`*. For example:

```
5.0`25 ^ 73
```

$$1.058791184067875423835 4 \times 10^{51}$$

As an optimization, approximate numbers that are input with no more digits than are supported by the computer's floating-point hardware are stored in native double-precision floating-point format, and all arithmetic operations on such numbers are performed in hardware. Numbers such as this are termed *machine-precision* numbers. The read-only system variable `$MachinePrecision` specifies the precision of "native" floating-point numbers, which may vary on different processor architectures.

This is the native floating-point precision supported by a PowerPC 601 processor.

```
$MachinePrecision
16
```

1. *Caveat:* The result of `Precision` sometimes does not match the number of digits that is displayed. The reason for this is that numerical precision actually is tracked in terms of binary digits, and is converted to the nearest number of decimal digits by `Precision`.

`MachineNumberQ[`*num*`]` can be used to determine if an approximate number is a machine number.

This input has too many digits to be a machine number.

```
MachineNumberQ[5.000000000000000000000000]
False
```

This input implicitly has `$MachinePrecision` significant digits.

```
MachineNumberQ[5.0]
True
```

This computation is done in the floating-point hardware.

```
5.0^73
1.05879 10^51
```

Like a pocket calculator, *Mathematica* displays only the first few digits of a machine-precision number, unless specifically asked for more. One way to see all of the digits is to use `FullForm`, but this would print the internal form of everything else in an expression as well. Alternatively, you can use `InputForm`, which asks *Mathematica* to display what would have to be typed as input to equal the given expression. In the following calculation, `%` is a special input form that stands for "the previous result."

Users of version 2.2 will see the notation `*10^51` in place of `*^51`.

```
InputForm[%]
1.058791184067876*^51
```

Note that the least-significant digit in this result is incorrect (compare it to earlier results in this section). Since the computation is being done in the computer's hardware floating-point unit, *Mathematica* cannot keep track of the true precision of the result. As far as it's concerned, the result still has `$MachinePrecision` significant digits:

```
Precision[%]
16
```

It is important to note that while the precision of a machine number is always `$MachinePrecision`, the converse is not necessarily true: If the result of an arbitrary-precision calculation happens to have `$MachinePrecision` or fewer significant digits, it is still stored internally as an arbitrary-precision number. Use `MachineNumberQ` to be certain.

Also note that an input containing a decimal point is *always* considered to be an approximate number, even when you "know" it isn't. For *Mathematica* to assume otherwise would be incorrect.

This result is only *approximately* zero (note the decimal point).

```
3/4 - 0.75
0.
```

If what was really meant in this case was "exactly three-fourths," then that is what should have been entered.

To summarize, there are two broad classes of numbers: exact (which includes not only integers but also rational numbers and complex numbers with exact coefficients) and approximate (whose members always contain a decimal point). The approximate numbers are further divided into two subclasses: machine precision and arbitrary precision.

Mathematica has an unusual convention for handling exact numbers. By default, *Mathematica* will never perform any numeric operation that would turn an exact expression into an approximate one. For example:

`Sqrt` is the built-in square root function.

```
Sqrt[3]
Sqrt[3]
```

Mathematica does not rewrite the exact expression `Sqrt[3]` as a number because attempting to do so would require an approximation — there is no finite-length decimal representation of the square root of 3. On the other hand, *Mathematica* will evaluate the following:

```
Sqrt[3.]
1.73205
```

The argument `3.` is considered to be approximate by virtue of its decimal point. Since the number is already approximate, *Mathematica* has no qualms about computing its square root.

You can ask *Mathematica* to numerically evaluate any exact expression by using the `N` function.

```
N[Sqrt[3]]
1.73205
```

By default, the result is a machine number.

```
InputForm[%]
1.732050807568877
```

An optional second argument to `N` specifies the desired precision of the result.

```
N[Sqrt[3], 90]
1.73205080756887729352744634150587236694280525381038062805580697945193301690880003708114618\
7
```

In general, if any of the arguments passed to a built-in numeric function are approximate, the function will evaluate, converting those arguments to approximate numbers with the highest precision that can be justified. For example,

This result is a machine number.

```
1 + 2.5
3.5
```

This result is an arbitrary precision number.

```
1 + 2.5000000000000000000000000000001
3.5000000000000000000000000000001
```

The foregoing discussion should not be interpreted to mean that *Mathematica* won't ever rewrite a numeric function of exact arguments. In fact, *Mathematica* includes rewrite rules for most special cases of the built-in numeric functions.

```
Sqrt[8]
2 Sqrt[2]
```

This rewriting has produced another exact answer; however, there is nothing more that can be done with the result that does not require an approximation.

Exact numeric expressions can be partitioned into two categories: those that are numbers and those that would become numbers if `N` were applied to them. The latter category includes not only expressions such as `Sqrt[3]` but also the built-in *numeric constants* `Catalan`, `Degree`, `E`, `EulerGamma`, `GoldenRatio`, and `Pi`. Although these are symbols, in certain circumstances they are treated as exact numbers. For example:

`Pi` is an exact representation of the irrational number π.

```
N[Pi]
3.14159
```

Therefore, this evaluation is exact.

```
Sin[Pi/3]
  Sqrt[3]
  -------
     2
```

Here's an example using `E`.

```
E^(a Log[x])
 a
x
```

You can think of numeric constants as symbols for which a large number of special-case rules are built into the system. They are not manifest numbers:

```
NumberQ[Pi]
False
```

— but they become numbers when `N` is applied to them.

Numeric constants are treated slightly differently in version 3.0 — they have become more "number-like." For example, in version 2.2 the following expressions would not be rewritten unless `N` were applied to them:


```
180. Degree
3.14159

Round[Pi]
3

EulerGamma + Catalan < GoldenRatio
True
```

A new function, `NumericQ`, can determine whether an expression represents a numeric quantity:

```
NumericQ[Pi]
True
```

The numeric property propagates through complicated expressions.

```
NumericQ[Sin[Pi]^(1/E)]
True
```

You can think of `NumericQ` as determining whether or not an expression would become numeric if `N` were applied to it. (The same information can be obtained in version 2.2 with `NumberQ[N[expr]]`, but this construct is considerably less efficient.)

An extensive discussion of numeric types and numerical precision can be found in §3.1 of *The Mathematica Book*.

Character strings

A *character string*, or simply *string*, is any sequence of characters enclosed in a pair of double quotes.

This is a string. Note that *Mathematica* does not print the quotes when it prints a string.

```
"hello world"
hello world
```

You can verify that the output is a string by requesting its `InputForm`.

```
InputForm[%]
"hello world"
```

Within a string, the sequence of characters `\"` stands for the single character ". Hence, the following is a valid string also.

The inner quotes *are* printed because they are just like any other characters in the string.

```
"For example, \"hello world\" is a string."
For example, "hello world" is a string.
```

Any 8-bit character can be entered directly into a string by using your system's special keyboard combination for that character. Alternatively, 8- and 16-bit characters can be entered by using various sequences of 7-bit ASCII characters; see §A.2.1 of *The Mathematica Book* for the exact syntax.

There are many built-in operations on character strings, such as finding their length, concatenating them, case-shifting them, and searching for and replacing substrings (including wildcards). These operations will be discussed in Section 3.6, "Character Strings."

Exercise

1. Determine how *Mathematica* stores complex numbers by using `FullForm` on several examples. Be sure to try symbolic as well as numeric coefficients.

2.2 Evaluation of Expressions

The basic evaluation process is very simple: The kernel continues to rewrite terms until there is nothing left that it knows how to rewrite in another form. Since term rewriting replaces one expression by another, whatever is left after this process terminates must be a valid expression, which implies that the set of all expressions is *closed* under evaluation. This allows any expression to be nested inside of any other (although the result of doing so may not make any sense!). By analogy with function call in other languages, we often will call the result of evaluating an expression the *return value* of the expression.

You can obtain a post mortem description of the evaluation of any expression by wrapping the expression inside the head `Trace`. The results may be slightly different, depending on the version of *Mathematica* that is being used.

```
Trace[Sin[Log[2.5, 7]]]
{{Log[2.5, 7], Log[7.]/Log[2.5],
   {{Log[2.5], 0.916291}, 1/0.916291, 1.09136},
   {Log[7.], 1.94591}, 1.09136 1.94591, 2.12368},
  Sin[2.12368], 0.851013}
```

```
Trace[Sin[Log[2.5, 7]]]
{{Log[2.5, 7], Log[7.]/Log[2.5], {Log[7.], 1.94591},
   {Log[2.5], 0.916291}, 1.94591/0.916291, 2.12368},
  Sin[2.12368], 0.851013}
```

The curly braces indicate the *depth* of the subexpressions being evaluated. The steps involved in this evaluation in version 3.0 are:

1. `Log[2.5, 7]` is rewritten as `Log[7.]/Log[2.5]`.
2. `Log[7.]` is evaluated numerically, yielding `1.94591`....
3. `Log[2.5]` is evaluated numerically, yielding `0.916291`....
4. The quotient of the two previous results is evaluated numerically.
5. The `Sin` of the quotient is evaluated numerically.
6. The result is an atom (a real number), which cannot be rewritten, hence the process terminates.

(In version 2.2, the two logarithms are evaluated in the reverse order and the division operation is replaced by a reciprocal and a multiplication.)

This example demonstrates an important point about the evaluation process: In general, the parts of a normal expression are evaluated before the entire expression is. This is called *standard evaluation*. (The arguments to certain *Mathematica* functions are *not* evaluated before the function is called; this is called *nonstandard* evaluation, an example of which will be encountered shortly.) In computer science terms, an expression is a *tree*, and standard evaluation is performed *depth first*. Figure 2-1 shows a tree representation of the evaluation performed above. Note that the nesting of the curly braces in the trace output (above) increases as the evaluation process moves deeper into the tree, and decreases as the evaluation process works its way back up.

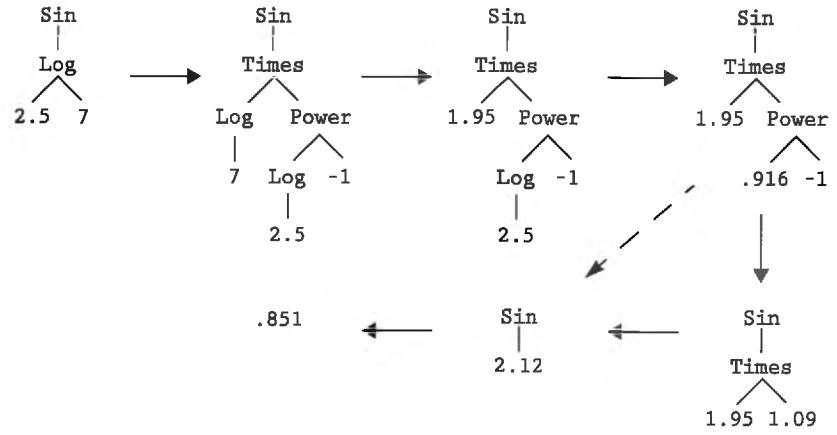


Figure 2-1 Tree representation of evaluation of the expression `Sin[Log[2.5, 7]]`. The process is streamlined somewhat in version 3.0 (as indicated by the dashed arrow).

Mathematica has a built-in function called `TreeForm` that attempts to print an expression in the form of a tree, subject to the limitations of ASCII output. Here is an example:

```
TreeForm[x + Cos[y]*Log[z]]
Plus[x, |                    ]
       Times[|      , |      ]
             Cos[y]  Log[z]
```

The output of `TreeForm` is not very appealing to look at, especially when the expression being formatted is so large that the output wraps across multiple lines. It is usually better just to examine the `FullForm` carefully.

Incidentally, look at what happens if we try to print the `TreeForm` of the example in Figure 2-1:

```
TreeForm[Sin[Log[2.5, 7]]]
0.851013
```

What happened is that the argument to `TreeForm` evaluated to a real number before `TreeForm` itself was evaluated. (The same problem would have happened with `FullForm` or `InputForm`.) In order to see the internal structure of such expressions, it is necessary to prevent them from evaluating. You can accomplish this by wrapping the expression inside of a head that does not allow its arguments to be evaluated. One such head is `Hold`:

`Hold` prevents the evaluation of its arguments, allowing us to see the structure of expressions without evaluating them.

```
TreeForm[Hold[Sin[Log[2.5, 7]]]]
Hold[|                ]
     Sin[|           ]
         Log[2.5, 7]
```

This is an example of *nonstandard evaluation*. You can tell that a symbolic head prevents the evaluation of its parts by examining its *attributes:*

The `HoldAll` attribute indicates that none of the parts enclosed by the head `Hold` will be evaluated.

```
Attributes[Hold]
{HoldAll, Protected}
```

We will cover `HoldAll` and related attributes in Section 4.4, "Parameter-Passing Semantics," and we will discuss techniques for operating on *held expressions* in Section 7.2, "Nonstandard Evaluation."

The head of an expression, too, can be an expression, and it is evaluated before any of the parts. Consider the following:

This input creates a rewrite rule for the expression `f[0]`.

```
f[0] = Sin
Sin
```

This expression has a nonatomic head.

```
Trace[f[0][Pi/2]]
{{f[0], Sin}, {{1/2, 1/2}, Pi/2, Pi/2}, Sin[Pi/2], 1}
```

The trace shows that the head of this expression, `f[0]`, evaluated to the symbol `Sin` before any parts were evaluated. The parts were then evaluated, and finally the overall expression (`Sin[Pi/2]`) was evaluated.

The trace is curious in that it contains what appear to be several trivial evaluations (`1/2` ⇒ `1/2`, `Pi/2` ⇒ `Pi/2`). In fact, these expressions are being rewritten from one internal form to another, which we can verify by looking at the `FullForm` of the trace output:

```
FullForm[%]
List[List[HoldForm[f[0]], HoldForm[Sin]],
  List[List[HoldForm[Power[2, -1]],
```

```
  HoldForm[Rational[1, 2]]],
 HoldForm[Times[Pi, Rational[1, 2]]],
 HoldForm[Times[Rational[1, 2], Pi]]],
HoldForm[Sin[Times[Rational[1, 2], Pi]]],
HoldForm[1]]
```

We can see from this output that the first instance of `1/2` was the normal expression `Power[2, -1]`, whereas the second instance was the number `Rational[1, 2]`. Since both of those expressions have the same output form, it appeared as though no rewriting was taking place. Similarly, the first instance of `Pi/2` was expressed internally as `Times[Pi, Rational[1, 2]]` (the `Rational` being the result of the previous rewrite). This expression was rewritten as `Times[Rational[1, 2], Pi]` because the arguments to a commutative operator like `Times` are sorted into canonical order before that operator is evaluated. (This makes algebraic simplification easier.)

Another curious feature of the trace output is the presence of the `HoldForm` head wrapped around every intermediate expression. `HoldForm`, like `Hold`, has the attribute `HoldAll`, which prevents the evaluation of its parts, but it differs from `Hold` in that in standard output form the head does not appear. The `HoldAll` attribute prevents the multiplication in `HoldForm[Times[Rational[1, 2], Pi]]` from being carried out, for example. As stated at the very beginning of this section, the evaluation process continues until there is nothing left that can be rewritten in another form. If not for the existence of heads like `Hold` and `HoldForm`, there would be no way for an expression to return a partially evaluated result such as a component of a trace.

At the beginning of this section we asserted that every expression returns another expression as its value. There are cases in which this seems to be untrue, however. An example is the `SetDelayed` operator, which doesn't seem to return a value:

The := operator is described in detail in Section 2.3.4.

```
s3 := Sqrt[3]
```

The above example seems to imply that there is no return value from `SetDelayed`. In fact, `SetDelayed` returns the special symbol `Null`, which normally doesn't appear in the output. `Null` will appear if it is part of a larger expression, however:

```
1 + %
1 + Null
```

`SetDelayed` is one example of a function that operates by producing a *side effect;* i.e., the intended result of executing the function is not its return value, but rather some change that it makes to the state of the *Mathematica* session (or to the computer in general — for example, writing data to a file). In this case, the side effect is that a rewrite rule for the symbol `s3` has been created:

`s3` is rewritten as `Sqrt[3]`.

```
s3^2
3
```

Other common side effects include the rendering of graphics and file input/output.

There is one last point that needs to be made before we leave this topic. An implicit assumption in the evaluation process is that the system is designed so that the set of all expressions is partially ordered with respect to evaluation, or equivalently, that there are no cyclic dependencies in the evaluation process. It is all too easy to construct an example for which this assumption is violated:

```
yin := yang
yang := yin
```

These statements instruct the kernel that `yin` can be rewritten as `yang` and that `yang` can be rewritten as `yin`. Obviously, this is a recipe for disaster! Fortunately, the kernel has a built-in "circuit breaker" known as the *iteration limit:*

```
yin
$IterationLimit::itlim:
   Iteration limit of 4096 exceeded.
Hold[yin]
```

After 4096 rewritings have taken place, the kernel wraps the current result in `Hold` (which prevents any further evaluation) and returns it. The fact that the final result is the same as the original expression is a quirk of the definitions used. We can examine this process in detail using `Trace`, but first we had better reduce the iteration limit to keep the output manageable:

The iteration limit can be changed by assigning to the global variable `$IterationLimit`.

```
$IterationLimit = 20
20

Trace[yin]
$IterationLimit::itlim:
   Iteration limit of 20 exceeded.
{yin, yang, yin, yang, yin, yang, yin, yang, yin,
  yang, yin, yang, yin, yang, yin, yang, yin, yang,
  yin, yang, yin, {Message[$IterationLimit::itlim,
    20], {$IterationLimit::itlim,
    Iteration limit of `1` exceeded.}, Null},
  Hold[yin]}
```

The trace shows that a game of "rewrite Ping-Pong" is being played between the symbols `yin` and `yang`. The `Message` function, which is invoked directly by the kernel and is not part of the original expression or any of its intermediate forms, is what causes the error message to appear. `Message` also can be called directly; its use is discussed in Section 9.1, "Diagnostic Messages."

Before we continue, it would be a good idea to restore the default value of `$IterationLimit`.

```
$IterationLimit = 4096;
```

This is all you need to know about the evaluation process for now; any further details would serve to confuse rather than to illuminate. We will return to the evaluation process in Chapter 7, "Expression Evaluation," after we have gained some programming experience.

Exercise

1. Failure to understand the evaluation process is the source of many common errors. For example, why doesn't *Mathematica* return a high-precision result from the following numerical computation?

```
N[Sqrt[3.], 90]
1.732050807568877
```

(The correct way to perform this computation was demonstrated in Section 2.1.2 on page 22.)

2.3 Special Input Forms

Now that we understand the basic structure of expressions and how they are evaluated, we turn our attention to the rich syntax that is available for entering expressions. Much of the syntax is evocative of other programming languages; the C language probably had the strongest influence on the syntax of *Mathematica*. However, identical syntax does not necessarily imply identical *semantics*, and we will take pains to point out pertinent differences as we go along.

We will not attempt to describe every special input form here, only the most common and elementary ones (see [Wolfram 91] §A.2.3 or [Wolfram 96] §A.2.7 for an exhaustive list). We will encounter additional special input forms throughout the remainder of the book.

2.3.1 Arithmetic operators

The arithmetic operators are + (addition), - (subtraction), * (multiplication), / (division), and ^ (exponentiation). Addition and subtraction have a lower precedence than multiplication and division, which in turn have a lower precedence than exponentiation.

This is `3 + 8`.

```
3 + 2 * 4
11
```

This is `3 * 16`.

```
3 * 2 ^ 4
48
```

When in doubt, you can substitute symbolic arguments and look at the `FullForm` of the expression.

```
FullForm[Hold[a * b ^ c]]
Hold[Times[a, Power[b, c]]]
```

You can use ordinary parentheses to override the precedence of operations; this is the *only* thing that parentheses are used for in *Mathematica*.

This is 6^4.

```
(3 * 2)^4
1296
```

Additive and multiplicative operators are left-associative, whereas exponentiation is right-associative.

Since multiplication and division are left-associative, the factor of 7 is in the numerator rather than in the denominator.

```
3 / 5 * 7
21
--
5
```

Since exponentiation is right-associative, 3^0 is evaluated first.

```
2 ^ 3 ^ 0
2
```

Multiplication is sometimes inferred even when no asterisk is present, as in `2a`. However, see Section 2.3.11, "Syntax traps for the unwary," for some caveats.

Note that although there are built-in functions called `Subtract` and `Divide`, these operations are converted immediately to additive and multiplicative inverses. For example,

```
FullForm[a - b]
Plus[a, Times[-1, b]]
```

Also note that `Plus` and `Times` are *n*-ary operators, i.e., they can take any number of arguments.

```
FullForm[a + b + c + d]
Plus[a, b, c, d]
```

One other common special input form that we will mention here is `n!`, which is equivalent to `Factorial[n]`. The factorial operator has a higher precedence than any of the arithmetic operators.

Exercise

1. Try to guess the internal forms of `b/c` and `c/b`. Verify your answer using `FullForm`.

2.3.2 Relational and boolean operators

Mathematica's relational and boolean operators are the same as C's, with one exception: there is no exclusive-or operator (but there is an `Xor` function). Here are some examples.

> is Greater; != is Unequal; && is And.

```
7 > 4 && 2 != 3
True
```

<= is LessEqual; || is Or; == is Equal.

```
7 <= 4 || 2 == 3
False
```

The boolean operators And and Or are *n*-ary.

As in C, the boolean operators "short circuit" once their outcome has been determined. For example, in the following expression the first argument to And evaluates to False, so the second argument is never evaluated:

1/0 normally would generate an error message.

```
1 < 0 && 1/0
False
```

A bit of thinking leads us to the conclusion that And (and Or) must undergo nonstandard evaluation — its arguments are evaluated under the control of the function, rather than before the function is called. (Verify this by checking its Attributes.)

There are a couple of differences in the usage of the relational operators from the way they are used in C. First of all, you can "chain" them.

This expression is the same as 5 > 4 && 4 > 3.

```
5 > 4 > 3
True
```

Second, and more important, the arguments to a relational operator need not evaluate to numbers, in which case it's possible that the relational operator will not evaluate.[2]

This may not be what you had in mind.

```
a == b
a == b
```

When comparing two symbolic expressions, you may want to test them for "sameness." This is accomplished by using the SameQ function, abbreviated === (triple equal sign).

Since a and b are not manifestly identical, SameQ returns False.

```
a === b
False
```

Mathematica knows about properties such as the associativity of addition.

```
a + b === b + a
True
```

The logical negation of SameQ is called UnsameQ, abbreviated =!=. Remember, SameQ and UnsameQ always evaluate to either True or False; Equal and Unequal may not.

---

2. In version 3.0, relational operators evaluate as long as both of their arguments are *numeric* (refer to the discussion of numeric expressions on page 24).

2.3.3 Reusing results

The % character stands for the result of the most recent evaluation. For example:

Here is an evaluation.

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

$$\frac{\text{Sqrt[3]}}{2}$$

% here refers to `Sqrt[3]/2`, the result of the last evaluation.

```
ArcSin[%]
```

$$\frac{\text{Pi}}{3}$$

(Incidentally, this example shows the benefit of working with exact quantities: Had either of the last two evaluations been done numerically, the eventual result would have been only an *approximation to* π/3.)

More generally, a sequence of *n* percent signs refers to the *n*th previous result.

```
ArcCos[%%]
```

$$\frac{\text{Pi}}{6}$$

You can also use `%`*n* or `Out[`*n*`]` to refer to output number *n*. However, this is a dangerous tactic, because if you save your notebook and reopen it during another *Mathematica* session, chances are extremely remote that these references will be correct. A much better strategy is to assign the results of computations to symbols, which we will see how to do in the next section.

Bear in mind that % always refers to the most recent output *in the order of evaluation*. Because the notebook interface allows you to move the insertion point around and to cut/copy/paste cells, the most recent output is not necessarily the same output cell that textually precedes the current input cell! Fortunately, the front end provides commands that copy the contents of the preceding input or output cell into the current input cell. These commands can be found in the **Prepare Input** submenu of the **Action** menu.

2.3.4 Assignment statements

The operators = (which is called `Set`[3]) and `:=` (which is called `SetDelayed`) perform a role in *Mathematica* that is similar to that of *assignment* in other languages. Either of the expressions *`lhs = rhs`* or *`lhs := rhs`* creates a rewrite rule for *`lhs`*. The difference between `Set` and `SetDelayed` is their treatment of *`rhs`*; this difference is subtle and will be illustrated by an example.

3. `Set` is a verb, not a noun. The `Set` operation has nothing to do with *sets* in the mathematical sense. Furthermore, mathematicians have to get used to the idea that = does not mean *equivalent*; == is used for that (see Section 2.3.2).

The expression *sym* = *expr* defines the value of the symbol *sym* to be the value of the expression *expr*. This action is a side effect; the return value of *sym* = *expr* is simply the value of *expr*.

```
In[]:= z = x + y
Out[]= x + y
```

Now z evaluates to x + y.

```
In[]:= z / 2
Out[]=  x + y
        -----
          2
```

The process of expression evaluation is iterative; after a substitution has been made, further substitutions may be performed on the new expression. For example, suppose that we give x the value 3.

```
In[]:= x = 3
Out[]= 3
```

Then after z evaluates to x + y, x will evaluate to 3:

```
In[]:= z / 2
Out[]=  3 + y
        -----
          2
```

Tracing the computation shows the sequence of evaluations explicitly.

```
In[]:= Trace[z / 2]
Out[]= {{z, x + y, {x, 3}, 3 + y}, {1/2, 1/2}, (3 + y)/2, (3 + y)/2}
```

The value of z is still x + y. We can verify this by looking at a trace like the one above, or we can inspect the definition of z directly using ?z:

Global` will be explained in Section 8.1, "Contexts."

```
In[]:= ?z
Global`z

z = x + y
```

The implication of this is that if the value of either x or y were to change, so would the result of evaluating z:

```
In[]:= x = Sqrt[3]
Out[]= Sqrt[3]

In[]:= z / 2
Out[]=  Sqrt[3] + y
        -----------
            2
```

Things would be different if we had created the definition for z *after* assigning a value to x or y. First we'll remove the existing definition for z, using Clear[z] (it's always a good idea to clear definitions that you don't need any longer):

`Clear` returns `Null`.

```
Clear[z]
```

`z` has no value after being cleared.

```
?z
Global`z
```

Now we `Set` `z` to `x + y` once again:

`Sqrt[3]` comes from evaluating `x`.

```
z = x + y
Sqrt[3] + y
```

The value of `z` is now `Sqrt[3] + y`.

```
?z
Global`z

z = Sqrt[3] + y
```

The crucial observation here is that the right-hand side of a `Set` is evaluated *before* any value is assigned to the left-hand side. Thus, the kernel substituted `Sqrt[3]` for `x`, performed the arithmetic, and the result, `Sqrt[3] + y`, became the new value of `z`.

It is quite commonly the case that one wishes to prevent this sort of thing from happening. That can be accomplished by making the assignment using `SetDelayed` rather than `Set`.

```
Clear[z]
z := x + y
```

The first thing we notice is that `SetDelayed` returns `Null`. This is because the expression on the right-hand side has not been evaluated. The value of `z` is the *literal* expression `x + y`, in spite of the fact that `x` has the value `Sqrt[3]`.

```
?z
Global`z

z := x + y
```

An important consequence of this is that subsequent changes to `x` or `y` will change how `z` evaluates.

Change the definition of `x`.

```
x = 2 Pi
2 Pi
```

`z` evaluates to `x + y`, which in turn evaluates `2 Pi + y`.

```
z
2 Pi + y
```

The example just given is a sort of "poor person's function definition." A much better way to do this is to declare `z` as a function of two *parameters*, `x` and `y`. We will demonstrate this in the next section.

C programmers will be glad to know that *Mathematica* has borrowed a few "hybrid" arithmetic/assignment operators from C. An expression like `x += y` is equivalent to

`x = x + y`. Analogous operators exist for subtraction (`-=`), multiplication (`*=`), and division (`/=`). There is no corresponding operator for exponentiation because the syntax `^=` is used for something entirely different.[4] Similarly, there are pre- and post-increment (`++`) and decrement (`--`) operators. `++x` is equivalent to `x += 1`: It adds `1` to `x`, stores the new value in `x`, and returns the new value. `x++` is similar except that it returns the *old* `value` of `x`. (If you have to ask why, then obviously you've never programmed in C.[5])

Be sure to clear x and z before continuing.

```
Clear[x, z]
```

Exercises

1. After setting `x` to 3 in your own *Mathematica* session, use `Trace` to trace the evaluation of the expressions `z := x + y` and `z = x + y`.
2. Given the following definitions (use `?Random` to find out what the `Random` function does, if you haven't already guessed):

```
x = Random[]
y := Random[]
```

   What is the result of evaluating the expressions `x - x` and `y - y`? To make certain you really understand what is going on, *evaluate these expressions more than once*.

2.3.5 Function call

As noted earlier, the syntax `head[parts...]` for constructing expressions can be thought of as a function call, and the resemblance to function call in other languages is no coincidence. Note that you must use square brackets, not parentheses, to indicate a function call. In fact, using parentheses is such a common mistake that the parser will even warn you about it:

This expression is equivalent to `Sqrt * 3`, which is meaningless.

```
Sqrt(3)
Syntax::bktwrn:
   Warning: "Sqrt(3)" should probably be "Sqrt[3]".
3 Sqrt
```

Believe it or not, there actually are three special input forms for function call — two for functions of a single argument and one for functions of two arguments. The first two

4. A description of which would be completely incomprehensible at this point. Wait until Section 6.5.2, "Upvalues."
5. The historic reason for these operators' presence in C is that they corresponded to assembly-language addressing modes on the DEC PDP-8. C programmers love them because they allow one to create the infamous one-line string copy function. The only explanation for their appearance in *Mathematica* is that they are the programming language equivalent of an inherited defect.

forms allow the function name to precede (*prefix* function application) or follow (*postfix* function application) the argument to the function. The third form allows the name of the function to appear in between its two arguments (*infix* function application).

Prefix function application is accomplished with the following syntax:

```
f @ x
f[x]
```

Prefix function application is popular among some users[6] because it eliminates the need for bracket matching, which can be quite a chore in a heavily nested expression. Of course, it works only for functions of a single argument. Prefix function application associates the way you would expect, that is, to the right.

```
f @ g @ h @ x
f[g[h[x]]]
```

Note that prefix function application has a very high precedence.

```
f @ x !
f[x]!
```

Postfix function application is accomplished by using the following syntax:

```
x // f
f[x]
```

Postfix function application commonly is used to apply a function that is peripherally related to the computation, and which might distract the reader from the main point if it appeared elsewhere. Examples of such uses are algebraic simplification, numerical approximation, and output formatting. Here's a simple example:

This is equivalent to `N[Sqrt[3]]` but is less distracting.

```
Sqrt[3] // N
1.73205
```

Postfix function application is left-associative and has a very low precedence.

```
x // f // g
g[f[x]]
```

```
a + b // f
f[a + b]
```

Infix function application is accomplished via the following syntax:

```
x ~ f ~ y
f[x, y]
```

6. But not with this author. You will rarely, if ever, see this syntax again in this book.

This notation is supposed to mimic the way the other binary operators (e.g., +) are used. For example, there is no special input form for logical exclusive or, but there is an `Xor` function:

```
True ~Xor~ False
True
```

The precedence of infix function application is just below that of prefix function application.

```
g @ x ~ f ~ y !
f[g[x], y]!
```

2.3.6 Function definition

Mathematica allows you to define your own functions.

This defines `z` as a function of two parameters, `x` and `y`.

```
z[x_, y_] := x + y
```

The expression on the left-hand side of the assignment is called the *declaration* of the function, and the expression on the right-hand side is called the *body* of the function. Whenever you use the function in an expression, the function is said to be *called*, and the value computed by the function (the *return value*) is substituted for the function call. When this function is called, for example, by evaluating

```
z[Sqrt[3], 2 Pi]
Sqrt[3] + 2 Pi
```

the values `Sqrt[3]` and `2 Pi` are substituted for `x` and `y`, respectively, everywhere that `x` and `y` occur in the body of the function. In computer science lingo, `x` and `y` are called *formal parameters*, and the values `Sqrt[3]` and `2 Pi` are called *actual parameters* or *arguments*.

Note that the values of the formal parameters `x` and `y` do not depend on the values of the global symbols `x` and `y`:

```
x := 1
y := Log[2]
z[Sqrt[3], 2 Pi]
Sqrt[3] + 2 Pi
```

or vice versa:

```
x + y
1 + Log[2]
```

Also note that we used `SetDelayed` to define the function. Had we used `Set`, the right-hand side would have been evaluated immediately, resulting in the substitution of any extant values for `x` or `y` into the body of the function. You should verify this.

The underscores in the left-hand side of the function definition are important: They indicate that `x` and `y` are formal parameters, as opposed to literal values (see Exercise 2.3.6.2). The underscore is called `Blank`, which is meant to suggest "fill in the blank." Note that the blanks appear only on the left-hand side of a function definition, never on the right-hand side.

Function definition is discussed in much greater detail in Part 2. For now, keep in mind that `SetDelayed` is almost always the right choice for defining a function, and that each formal parameter should be followed by a `Blank`.

Exercises

1. *The importance of using `SetDelayed`.* Clear `z` (but not `x` or `y`) and evaluate the following expressions (note the use of `Set` rather than `SetDelayed` in the function definition):

```
z[x_, y_] = x + y
z[Sqrt[3], 2 Pi]
```

2. *The significance of `Blank`.* Define a function (incorrectly) as follows:

```
Clear[f]
f[a] := a^2
```

   Now evaluate the expressions `f[a]` and `f[b]`. Do you understand the significance of the `Blank` now? Clear `f` and redefine it correctly. Test it to make sure!

3. Write a function that computes the area of a circle given its radius. (Use the built-in numeric constant `Pi` in your answer.)

Clear these symbols before continuing.

```
Clear[f, x, y, z]
```

2.3.7 Compound expressions

Multiple expressions can be placed anywhere a single expression could go by separating them with semicolons.

```
a = 1; b = 2; a + b
3
```

Note that the output of every expression that is followed by a semicolon is suppressed. This is quite handy for suppressing the output of trivial operations, or of expressions that evaluate to enormous messes.

An input such as the one above is actually a single expression; each of the individual "statements" is a part of an expression having the head `CompoundExpression`. This head is fairly elusive, in that you can't see it by wrapping it in FullForm:

```
FullForm[a = 1; b = 2; a + b]
3
```

The reason for this is — you should know this by now — the `CompoundExpression` evaluates before `FullForm` does. A way around this problem is to prevent the evaluation of the `CompoundExpression` by wrapping it in `Hold` (see the discussion of nonstandard evaluation in Section 2.2).

`Hold` prevents the `CompoundExpression` from evaluating.

```
FullForm[Hold[a = 1; b = 2; a + b]]
Hold[CompoundExpression[Set[a, 1], Set[b, 2],
   Plus[a, b]]]
```

Since `CompoundExpression` is a single expression, a sequence of expressions separated by semicolons can be placed anywhere a single expression could be used. The obvious utility of this is that it allows one to program in a style that is reminiscent of procedural languages such as C and Fortran. For example, a function that consists of more than one "line of code" can be written like this:

```
f[x_] := (
    firstLine;
    secondLine;
    ...
    lastLine
)
```

Take note of the parentheses surrounding the body of the function above. They are necessary because ; has the lowest precedence of any *Mathematica* special input form. Without the parentheses, only *`firstLine`* would be part of the function definition. Also note the absence of a semicolon after *`lastLine`*. A multiline function returns the value of the last expression evaluated by the function. If a semicolon appeared at the end of the last line, the function would return `Null`.

There are unobvious applications of compound expressions as well. For example, the `FindRoot` function searches for the root of an expression, given an initial "guess."

The form of the return value in this example will be explained in Section 2.3.9.

```
FindRoot[Sin[x] - Cos[x], {x, .5}]
{x -> 0.785398}
```

The answer gives no indication as to how `FindRoot` "got there." However, the sequence of iterates that are generated by `FindRoot`'s internal algorithm can be printed using the following technique:

The expression whose root is being sought is `Print[x]; Sin[x] - Cos[x]`. Thus, each time `FindRoot` evaluates this expression, the current value of `x` is printed.

```
FindRoot[Print[x]; Sin[x]-Cos[x], {x, .5}]
x
0.5
0.793408
0.785398
0.785398
{x -> 0.785398}
```

Clear `a` and `b` before continuing.

```
Clear[a, b]
```

Exercises

1. What is the internal representation of the following expression? (You will need to use the `Hold` trick to see it.)

```
z = 1;
```

2. *Mathematica*'s two-dimensional plotting commands use an interesting adaptive sampling algorithm (see [Smith & Blachman 94c] or [Wickham-Jones 94] for an explanation of the algorithm). You can find out *which* points were sampled by examining the internal form of the `Graphics` object generated by the plotting command, e.g.,

```
Plot[Sin[x], {x, Pi/4, Pi/2}] // InputForm
(* large output omitted *)
```

However, you can't determine the *order* in which the points were sampled from the `Graphics` object because the points are sorted before the object is returned. Use the same trick that we used in the `FindRoot` example to observe the sequence of sample points as the algorithm executes.

2.3.8 Lists

Lists are the basic data structure in *Mathematica*. A list is used to group expressions in a particular order. Curly braces delimit a list:

This is a list of three expressions.

```
{1, x, i + j}
{1, x, i + j}
```

The internal form of a list is quite straightforward.

```
FullForm[%]
List[1, x, Plus[i, j]]
```

The head `List` doesn't cause any evaluation to take place; it simply has appealing special input and output forms.

Many of the built-in functions require that certain parameters be grouped into lists. For example, in the expression `Plot3D[Sin[x y], {x, 0, Pi}, {y, 0, Pi}]`, the lists are used to group each independent variable with its desired plot range.

There is another set of delimiters that one encounters frequently when working with lists: double square brackets, which are used to extract the parts of a list.

This returns the third part of the list given above.

```
%[[3]]
i + j
```

Lists are therefore analogous to *arrays* in procedural programming languages such as C and Fortran. However, note that lists always use 1-based indexing, as in Fortran. (C programmers take note!)

Since lists are expressions, they can be nested arbitrarily. By convention, rectangular nested lists are used to represent matrices, with each row of the matrix stored as a sublist.

s is a nested list that represents a 3 × 2 matrix.

```
s = {{a, b}, {c, d}, {e, f}};
```

The elements of s also are lists; this represents a row of the matrix.

```
s[[1]]
{a, b}
```

The sublists can of course be indexed as well; this represents an element of the matrix.

```
%[[2]]
b
```

All of the above can be accomplished in a single step.

```
s[[1, 2]]
b
```

Lists are pervasive in *Mathematica;* in fact, there are so many built-in functions for operating on them that we will devote nearly all of Chapter 3 and a good part of Chapter 5 to them.

Exercise

1. What subscript is needed to extract the element c from the following list? How about e?

```
{a, {b, {c, d}, e}, f}
```

2.3.9 Rules

The special input form `a -> b` is called a *rule*. By itself, a rule is little more than a container for a pair of expressions (with a special input/output form), but there are several functions that expect rules as arguments. The most common such function is called `ReplaceAll`. `ReplaceAll[expr, a -> b]` replaces every occurrence of *a* in *expr* by *b*:

```
ReplaceAll[x + x^2, x -> w]
w + w^2
```

`ReplaceAll` is so common that there is a special input form for it: *`expr /. rule`* is equivalent to `ReplaceAll[`*`expr, rule`*`]`.

```
x + x^2 /. x -> w
w + w^2
```

`ReplaceAll` can replace an arbitrary expression by another arbitrary expression. Note, however, that rule substitution is purely syntactic, not algebraic:

`ReplaceAll` does not attempt to substitute for `x^4`, only for `x^2`.

```
x^2 + x^4 /. x^2 -> w + 1
1 + w + x^4
```

Here are some other properties of `ReplaceAll`:

The second argument to `ReplaceAll` can be a list of rules.

```
x^2 + x^4 /. {x^2 -> w + 1, x^4 -> (w + 1)^2}
1 + w + (1 + w)^2
```

The "`/.`" operator is left-associative.

```
x + x^2 /. x -> w^2 /. w^2 -> z
w^4 + z
```

Another common use for rules is for specifying *options* to a function, which are *named arguments* of the form *`name->value`*. (This is in contrast to ordinary *positional* arguments, whose names are inferred from their position in the sequence of arguments to the function.) Since options carry their names with them, they don't have to appear in any particular order — in fact, they don't have to appear at all. Options are thus useful for functions that take a lot of parameters (such as the graphics functions), or for parameters that are used infrequently. Options are always given after all positional arguments.

You can get a list of the options for a function (and their default values) by using `Options[`*`symbol`*`]`.

These are the options for the built-in function `FactorInteger`.

```
Options[FactorInteger]
{FactorComplete -> True, GaussianIntegers -> False}
```

397 cannot be factored over the ordinary integers.

```
FactorInteger[397]
{{397, 1}}
```

But it can be factored over the Gaussian integers.

```
FactorInteger[397, GaussianIntegers -> True]
{{-I, 1}, {6 + 19 I, 1}, {19 + 6 I, 1}}
```

If you make a lot of calls to a function with the same option specifications, you may find it advantageous to change the default options for the function. You can do this with `SetOptions`:

`SetOptions` returns the new set of default options.

```
SetOptions[FactorInteger, GaussianIntegers -> True]
{FactorComplete -> True, GaussianIntegers -> True}
```

Now `FactorInteger` behaves as though `GaussianIntegers -> True` were specified on every call.

```
FactorInteger[397]
{{-I, 1}, {6 + 19 I, 1}, {19 + 6 I, 1}}
```

2.3.10 Control flow

Although *Mathematica* contains functions for conditional execution such as `If` and `Switch`, and loop-control functions such as `Do`, `While`, and `For`, there is no special syntax for any of these. The reason for this is that they aren't used all that frequently. We'll discuss these functions in Chapter 4, "Procedural Programming."

2.3.11 Syntax traps for the unwary

There are a couple of features of the *Mathematica* parser that are intended to make input more natural for users, but can result in unintended consequences if you are not aware of them.

First of all, a space may be used instead of an asterisk to signify multiplication. This looks much like standard mathematical notation when symbols are being multiplied together:

```
a b
a b
```

In very special cases, even the space is unnecessary. The author knows of only two such cases. The first is when a number is followed by a nonnumeric character:

Since symbol names cannot begin with a digit, this is interpreted as multiplication.

```
2a
2 a
```

The second is when two symbols are separated by a delimiter such as a parenthesis or a list brace:[7]

The parenthesis between the symbols `a` and `b` implies multiplication.

```
a(b + c)
a (b + c)
```

7. This works for either left or right parentheses and list braces. It also works for single and double square brackets, but only on the right, of course, since `a[b]` is a function call and `a[[b]]` is a subscripting operation.

Note that *Mathematica*'s output always shows a space between symbols that are considered distinct, even if you leave the space out of the input. Conversely, if you *don't* see a space, *Mathematica* doesn't consider them to be distinct symbols.

Mathematica interpreted the two characters as a single symbol called `ab`.

```
ab
ab
```

Here's proof.

```
{a b / a, ab / a}
```

$$\{b, \frac{ab}{a}\}$$

This is a frequent source of errors. As a general rule, it's a good idea to use spaces (or parentheses) even when it's not strictly necessary; this improves readability in addition to avoiding errors.

The second feature that can entrap users, which is specific to the notebook interface, is the ability to enter multiple expressions in a single input cell. The problem is that if you have an expression that doesn't all fit on a single input line, *Mathematica* may try to interpret it as multiple expressions.

This is interpreted as two separate inputs. The second line causes a parse error.

```
3 * 4 + 5
* 2
17

Syntax::sntxb: Expression cannot begin with "* 2".
```

If you are really unlucky, you might not even get an error message. Here, the leading + is interpreted as a unary operator.

```
3 * 4
+ 5 * 2
12

10
```

The way to avoid this problem is to make sure that you never end an intermediate line of input in any way that might allow *Mathematica* to construe that line as a complete expression (unless it really is, of course).

Here is how to fix the example shown above.

```
3 * 4 +
5 * 2
22
```

If there are unmatched delimiters (parentheses, square brackets, or list braces) at the end of a line, *Mathematica* will not consider the expression to be complete.

```
Sqrt[3 * 4
+ 5 * 2]
Sqrt[22]
```

Attempting to end an input cell with some delimiters unmatched will, of course, result in an error message.

```
Sqrt[3 * 4
Syntax::sntxi: Incomplete expression.
```

Exercises

1. How does *Mathematica* interpret each of the following expressions, and why?

```
a^bc
a^2bc
```

2. What is the result of evaluating the following input in a single input cell?

```
{2 + 3,
 4 + 5
 6 + 7
}
```

Do you understand what happened?

2.3.12 Symbol information

Information about any symbol can be obtained with the syntax ?*name*.

Here's some information about the `FactorInteger` function.

```
?FactorInteger
FactorInteger[n] gives a list of the prime factors of
   the integer n, together with their exponents.
```

You can get even more detail using ??*name*.

```
??FactorInteger
FactorInteger[n] gives a list of the prime factors of
   the integer n, together with their exponents.

Attributes[FactorInteger] = {Listable, Protected}

Options[FactorInteger] =
  {FactorComplete -> True, GaussianIntegers -> False}
```

You can get information about any symbol — not just function names.

```
?GaussianIntegers
GaussianIntegers is an option for FactorInteger,
   PrimeQ, Factor and related functions which
   specifies whether factorization should be done over
   Gaussian integers.
```

The asterisk can be used to do wildcard searches for symbol names.

Here are the names of all functions that contain the word `Solve`. (Not all of these appear in version 2.2.)

```
?*Solve*
DSolve              NDSolve             SolveAlways
DSolveConstants     NSolve              SolveDelayed
LinearSolve         Solve               $DSolveIntegrals
MainSolve
```

Note that in version 3.0, ?*name* and ??*name* are expressions that return `Null` (the information that appears on the screen is printed, rather than being returned as a value).

However, in version 2.2 `?name` and `??name` are *not* expressions, and the `?` character will cause an error unless it is the first character on an input line.


```
        ?FactorInteger
Syntax::sntxf:
   "       " cannot be followed by "?FactorInteger".
```

The Mathematica Book calls this type of syntax *special input*. Other special input sequences are `!command` (executes an external command[8]) and `!!file` (displays the contents of a file). These last two must appear at the beginning of a line, even in version 3.0, or else the exclamation character is interpreted as the logical negation function `Not`. You might think of special input as being analogous to the # in C or the *continuation character* in Fortran.

Exercises

1. Find all system-defined symbols containing the word `Plot`.
2. If you have version 3.0, determine the internal representation of the expressions `?name` and `??name`. (Users of version 2.2 should skip this exercise.)

8. This feature is operating-system-dependent.

3

Lists and Strings

Lists are the basic data structure used in *Mathematica* programs. In addition to extracting parts of a list, you can Append, `Prepend`, `Insert`, or `Delete` parts to/from a list; `Take` or `Drop` an arbitrary number of parts from the front, back, or middle; `Rotate` it left or right; `Sort` it; `Reverse` it; "wrap a function around" every part in it (`Map`); `Count` parts that match a specification; and more. Operations on multiple lists include `Join` (concatenation), `Union`, `Intersection`, and `Complement`. In this chapter we will cover the basic repertoire of list operations; even more sophisticated operations on lists will be discussed in Chapter 5, "Functional Programming."

We conclude this chapter with a brief discussion of string operations, because they are in many ways analogous to list operations.

3.1 List Basics

3.1.1 List input and output

A list is composed of a sequence of expressions, separated by commas and enclosed in curly braces.

Here is a list of three expressions.

```
In[1]:= list1 = {a, N[Pi], Sqrt[3]}
Out[1]= {a, 3.14159, Sqrt[3]}
```

Since each part of a list is an expression, as is an entire list, lists can contain other lists.

Here is a nested list whose parts are also lists. The second part of the second sublist is also a list.

```
In[2]:= list2 = {list1, {d, {e, f}}}
Out[2]= {{a, 3.14159, Sqrt[3]}, {d, {e, f}}}
```

The `TableForm` output format prints a list in tabular form.

```
TableForm[list2]
a   3.14159   Sqrt[3]

d   e   f
```

Rectangular nested lists (i.e., lists in which the sublists at any given level of nesting have the same length) are used to represent matrices (in row-major form).

```
mat = {{a11, a12, a13}, {a21, a22, a23}};
```

The `MatrixForm` output format prints a list as a matrix, when possible.

```
MatrixForm[mat]
a11   a12   a13
a21   a22   a23
```

Note that a rectangular nested list could be formatted using either `MatrixForm` or `TableForm`. The difference between the two is that `MatrixForm` pads the columns of the output so that they have equal width, whereas `TableForm` does not.

```
TableForm[{list1, {d, e, f}}]
a   3.14159   Sqrt[3]
d   e         f

MatrixForm[{list1, {d, e, f}}]
a         3.14159   Sqrt[3]
d         e         f
```

Both formatting commands accept options that control item alignment, row and column spacing, table headings, and so forth. See [Wolfram 91] §2.7.6 or [Wolfram 96] §2.8.8 for specifics.

3.1.2 Extracting parts of lists

To extract a part from a list, use *double* square brackets. The parts are numbered starting with 1. The time required to extract any part is independent of the part being accessed.

```
list1[[3]]
Sqrt[3]
```

You can extract multiple parts at a time by using a list of indices. The result is a list of items, in an order corresponding to the indices.

```
list1[[{3, 2}]]
{Sqrt[3], 3.14159}
```

Extracting a part from a nested list can be accomplished by using a comma-separated sequence of indices inside the double square brackets.

This expression is equivalent to `list2[[1]][[2]]`.

```
list2[[1, 2]]
3.14159
```

Since matrices are stored by rows, index *i* extracts the *i*th row.

```
mat[[2]]
{a21, a22, a23}
```

You can transpose the rows and columns of a matrix.

```
Transpose[mat] // MatrixForm
a11   a21
a12   a22
a13   a23
```

To get the *i*th column, extract the *i*th row of the transpose.

```
Transpose[mat][[2]]
{a12, a22}
```

To extract a submatrix consisting of rows *r1*, *r2*, ... and columns *c1*, *c2*, ..., use an index of the form `[[{r1, r2, ...}, {c1, c2, ...}]]`.

```
mat[[{1, 2}, {1, 3}]] // MatrixForm
a11   a13
a21   a23
```

The internal form of the expression `s[[i, j, ...]]` is `Part[s, i, j, ...]`. A difficulty arises when one has a list of integers that one wants to use as a single, multidimensional index. As demonstrated above, passing a list of integers to `Part` is not the same as passing the integers themselves — a list of integers is interpreted as multiple, one-dimensional indices. For example,

This extracts part 2 followed by part 1 from `mat`.

```
subs = {2, 1};
mat[[subs]]
{{a21, a22, a23}, {a11, a12, a13}}
```

In version 2.2 there are several ways around this problem that involve the use of some fairly sophisticated programming techniques (see page 197). In version 3.0, however, there is a new function, `Extract`, that is tailor-made for this problem. `Extract` is similar to `Part` except that the indices are specified in a list:


```
Extract[mat, subs]
a21
```

Furthermore, `Extract` allows you to specify multiple, multidimensional indices.

There is no way to do this using a single `Part` command.

```
Extract[mat, {{1, 2}, {2, 1}}]
{a12, a21}
```

`Extract` has additional features that we will explore in Section 7.3.2, "Part extraction and replacement."

Exercises

1. What happens if the list passed to `MatrixForm` is not rectangular?
2. Extract the symbol `xyz` from the list `{a, {b, c}, {{x, {xyz, y}, z}}}` using indexing.
3. Given a permutation of the integers `1` through `n` as a list, e.g., `{2, 1, 3, 5, 4}`, show how to permute any other list of length `n` into this order. In other words, turn `{a, b, c, d, e}` into `{b, a, c, e, d}`.

3.2 Generating Lists

Entering a list directly is fine for small lists, but for large lists some kind of automation is a necessity. There are three general-purpose automated list generators:[1] `Range`, `Array`, and `Table`. In addition, there are a few special-purpose generators for matrices.

3.2.1 Range

`Range` can be used to generate a list of numbers in an arithmetic progression.

In its simplest form, `Range[n]` generates a sequence of the positive integers that are less than or equal to `n`.

```
Range[5]
{1, 2, 3, 4, 5}
```

If you pass two arguments to `Range`, they are taken as the starting and ending points of the sequence.

```
Range[4, 12]
{4, 5, 6, 7, 8, 9, 10, 11, 12}
```

A third argument specifies the step size.

```
Range[4, 12, 2]
{4, 6, 8, 10, 12}
```

Note that the range specifications do not have to be integers.

```
Range[2/3, 5.5, 9/11]
{2/3, 49/33, 76/33, 103/33, 130/33, 157/33}
```

3.2.2 Array

`Array[f, n]` or `Array[f, {n}]`, where `f` is a function of a single variable, creates the list `{f[1], ..., f[n]}`.

---

1. In addition to the operations discussed in this section, you can also read data from a file directly into a list using the `ReadList` function (see Section 12.4, "High-Level Input").

Here is a table of the first 10 prime numbers.

```
Array[Prime, 10]
{2, 3, 5, 7, 11, 13, 17, 19, 23, 29}
```

Given a function of multiple arguments and multiple bounds, `Array` will generate a matrix. In this case the list braces around the second argument are *not* optional.

```
Array[Less, {3, 4}]
{{False, True, True, True},
  {False, False, True, True},
  {False, False, False, True}}
```

It is easier to understand what is going on if we use a symbolic head that doesn't evaluate.

```
Array[f, {3, 4}]
{{f[1, 1], f[1, 2], f[1, 3], f[1, 4]},
  {f[2, 1], f[2, 2], f[2, 3], f[2, 4]},
  {f[3, 1], f[3, 2], f[3, 3], f[3, 4]}}
```

An optional third argument to Array specifies the base for the index (or indices). Compare the next input to the previous one to see why multiple indices must be enclosed in list braces.

```
Array[f, 3, 4]
{f[4], f[5], f[6]}
```

3.2.3 Table

`Table` is the most general of the iterators. It takes a minimum of two arguments: an expression to be iterated and one or more range specifications of the form {*var*, *start*, *end*, *step*}. The result is a list of the expression iterated over the given range(s).

Here's another way to generate the table of primes.

```
Table[Prime[i], {i, 1, 10, 1}]
{2, 3, 5, 7, 11, 13, 17, 19, 23, 29}
```

A range specification with only three parts is interpreted as {*var*, *start*, *end*, 1}.

```
Table[x^2 - 4 x, {x, 3, 7}]
{-3, 0, 5, 12, 21}
```

Note that the first argument to `Table` is an expression that depends on an explicit iteration variable, which is specified in the second argument. Contrast this with `Array`, in which the iteration variable is implicit.

If there is more than one iterator, the resulting list is multidimensional. The last iterator varies most quickly.

A range specification with only two parts is interpreted as {*var*, 1, *end*, 1}.

```
Table[a[i, j], {i, 2}, {j, 2, 4}]
{{a[1, 2], a[1, 3], a[1, 4]},
  {a[2, 2], a[2, 3], a[2, 4]}}
```

The range specification can even omit the variable name in cases where the name does not appear inside the expression to be iterated.

```
Table[Random[], {3}]
{0.312451, 0.0554533, 0.767995}
```

3.2.4 Generating matrices

In addition to the general-purpose list-generating commands just presented, there are two special generators just for matrices:

```
IdentityMatrix[4]
{{1, 0, 0, 0}, {0, 1, 0, 0}, {0, 0, 1, 0}, {0, 0, 0, 1}}

DiagonalMatrix[{a, b, c, d}]
{{a, 0, 0, 0}, {0, b, 0, 0}, {0, 0, c, 0}, {0, 0, 0, d}}
```

Exercises

1. Generate a list of (a) the first 10 integers, (b) all integers between 30 and 40, (c) all of the odd integers between 30 and 40.
2. Generate a list of values of the `BesselJ[0, x]` function at the points `1, 1.5, 2, 2.5, ..., 10`.
3. Create a 2-D list of values `Binomial[i + j, i]` for integer values of `i` and `j` running from 0 (not 1) up to some integer. Display this list using `TableForm`.
4. Modify the answer to the previous exercise so that the table takes on an upper-triangular form. (*Hint:* You can make the upper limit for `j` depend on `i`.) Do you see a well-known pattern?

3.3 Listable Functions

Virtually all of the built-in numerical functions (and a few symbolic ones as well) can take lists as arguments, which they operate upon element by element. This feature is called *listability*. For example, the following expression takes the square root of each element of the list.

```
Sqrt[{a, b, c}]
{Sqrt[a], Sqrt[b], Sqrt[c]}
```

Contrast the above with `Sqrt[a, b, c]`, which would generate an error message.

If the function takes multiple arguments, then zero or more of the arguments can be lists, so long as all of the list arguments are the same length.

Here the first argument to Power is a list.

```
{a, b, c} ^ d
{a^d, b^d, c^d}
```

Here the second argument is a list.

```
a ^ {d, e, f}
{a^d, a^e, a^f}
```

And here both arguments are lists. The function is applied element by element.

```
{a, b, c} ^ {d, e, f}
{a^d, b^e, c^f}
```

This fails because the lists are not the same length.

```
{a, b, c} ^ {d, e}
Thread::tdlen:
                                                    {d, e}
   Objects of unequal length in {a, b, c}
     cannot be combined.
                  {d, e}
{a, b, c}
```

Listability makes it possible to do computations on entire lists as easily as on single elements (i.e., *vector arithmetic*). Note that although lists can be thought of as vectors, multiplying two lists results in an element by element multiplication rather than a dot product. To do otherwise would not be consistent with the way the other arithmetic operators work on lists.

This is element-wise multiplication.

```
{a, b, c} {d, e, f}
{a d, b e, c f}
```

Use the "." operator if you want a dot product.

```
{a, b, c} . {d, e, f}
a d + b e + c f
```

Listable functions also work on matrices, albeit in a two-step process that normally is invisible. We can expose the inner workings of this process using `Trace`:

First the `Plus` operation is "pushed inside" the matrix to operate on each row; then it is "pushed inside" each row, eventually operating on each element of each row.

```
Trace[{{x11, x12}, {x21, x22}} + y]
{{{x11, x12}, {x21, x22}} + y,
  {{x11, x12} + y, {x21, x22} + y},
  {{x11, x12} + y, {x11 + y, x12 + y}},
  {{x21, x22} + y, {x21 + y, x22 + y}},
  {{x11 + y, x12 + y}, {x21 + y, x22 + y}}}
```

The same mechanism allows one matrix to be added to another (having the same dimensions, of course); you should trace a small example to see it at work.

Even assignment is listable, which is very handy indeed:

This assigns 1 to `x` and 2 to `y`.

```
{x, y} = {1, 2}
{1, 2}
```

Here is a one-liner that swaps the values of `x` and `y`.

```
{x, y} = {y, x};
{x, y}
{2, 1}
```

To find out if a function is listable, check its *attributes*.

`Sqrt` is a listable function.

```
Attributes[Sqrt]
{Listable, NumericFunction, Protected}
```

You can make your own functions listable simply by adding this attribute to them.

This function is not listable.

```
greater2[x_] := x > 2
greater2[{1, 2, 3}]
{1, 2, 3} > 2
```

Now the `greater2` function will work on list arguments.

```
SetAttributes[greater2, Listable]
greater2[{1, 2, 3}]
{False, False, True}
```

Just be certain that it makes sense to do so. For example, when the first argument to `Plot` is a list, the function produces a single plot with several curves on it, not several individual plots. If the `Plot` function were `Listable`, it could never "see" a list argument and this behavior would be impossible.

Caveat: `Clear[f]` removes definitions for `f`, but does not remove attributes. Use `ClearAll[f]` to do both.

All of the built-in numerical functions (e.g., `Plus`, `Sin`, `Gamma`), predicates (e.g., `IntegerQ`, `PrimeQ`), and some symbolic functions (e.g., `Together`, `ToExpression`) are listable. An inventory of *Mathematica*'s listable functions can be found in Table 3-1 on page 67. See Exercise 5.2.1.3 to learn how that table was generated.

Exercises

1. Generate a list of the square roots of all the odd integers between 1 and 20 without using `Table` or `Array`.
2. Given a list of coefficients of a polynomial (from low order to high order, including explicit zeros), generate that polynomial using `Table` and `Dot`.

3.4 Getting Information about Lists

There are several functions for querying the structure or contents of a list.

You can find the length of a list.

```
Length[{a, b, c}]
3
```

You can obtain the dimensions of a matrix. The length of this list is 2.

```
Dimensions[{{a, b, c}, {d, e, f}}]
{2, 3}
```

The `MemberQ` function tests a list for membership of a given expression.

```
{MemberQ[{a, b, c}, a], MemberQ[{1, 2, 3}, a]}
{True, False}
```

The `Count` function returns the number of expressions in the list that match a test expression.

```
{Count[{a, b, a, c}, a], Count[{a, b, a, c}, d]}
{2, 0}
```

The `Position` function returns the positions of the matches.[2]

```
Position[{a, b, {a, c}}, a]
{{1}, {3, 1}}
```

A *predicate* is a function that tests an expression for some property and always returns either `True` or `False` (for example, `MemberQ`). *Mathematica* contains many built-in predicates. Most functions ending in the letter `Q` are predicates, and vice versa. (The exceptions to this rule are `EllipticNomeQ`, `HypergeometricPFQ`, `InverseEllipticNomeQ`, `LegendreQ`, and `PartitionsQ`, which are not predicates.)

This displays the names of all functions that end in the letter `Q`. Nearly all of these are predicates. (Some of these appear only in version 3.0 or later.)

```
?*Q
ArgumentCountQ          MatrixQ
AtomQ                   MemberQ
DigitQ                  NameQ
EllipticNomeQ           NumberQ
EvenQ                   NumericQ
ExactNumberQ            OddQ
FreeQ                   OptionQ
HypergeometricPFQ       OrderedQ
InexactNumberQ          PartitionsQ
IntegerQ                PolynomialQ
IntervalMemberQ         PrimeQ
InverseEllipticNomeQ    SameQ
LegendreQ               StringMatchQ
LetterQ                 StringQ
LinkConnectedQ          SyntaxQ
LinkReadyQ              TrueQ
ListQ                   UnsameQ
LowerCaseQ              UpperCaseQ
MachineNumberQ          ValueQ
MatchLocalNameQ         VectorQ
MatchQ
```

Mathematica contains several functions that allow you to test the expressions in a list with some predicate.

`Select` picks out expressions in a list that satisfy a predicate.

```
Select[{a, 3, b, 2.7, 5}, IntegerQ]
{3, 5}
```

2. Note that the form of the return value of `Position` is suitable for input to some other functions, such as `Extract` (Section 3.1.2) and `ReplacePart` (Section 3.5.1).

`VectorQ` tests a list to see if it is a vector (i.e., a one-dimensional list).

```
{VectorQ[{a, b, c}], VectorQ[{a, {b, c}}]}
{True, False}
```

An optional second argument to `VectorQ` is a predicate that every element of the vector must satisfy.

```
{VectorQ[{a, b, c}, IntegerQ],
    VectorQ[{1, 2, 3}, IntegerQ]}
{False, True}
```

`MatrixQ` is the analog of `VectorQ` for matrices.

```
{MatrixQ[{{a, b}, {c, d}}],
    MatrixQ[{{a, b}, {c, {d, e}}}]}
{True, False}
```

Incidentally, there are four more functions that test their argument for a property but are not, strictly speaking, predicates: `Positive`, `NonPositive`, `Negative` and `NonNegative`. These are not true predicates because if their argument is nonnumeric, they simply do not evaluate.

```
Positive[a]
Positive[a]
```

Exercises

1. What happens when `Dimensions` is applied to a list that is not a matrix?
2. Write a one-line expression that tests a list to see if all of its entries are odd integers.
3. Write a one-line expression that tests a list to see if it is a matrix of negative numbers.
4. Write a one-line expression that tests a list to see if all of its parts are identical. (*Hint:* Find the number of parts in the list that are identical to the first one.)

3.5 Manipulating Lists

All of the operations discussed in this section modify a *copy* of the original list and return the copy, leaving the original list unchanged. Only two operations (`AppendTo` and `PrependTo`) overwrite the original list with the new list. This means you can experiment almost endlessly on a data set without worrying about overwriting it; on the other hand, it also means that the memory used by your *Mathematica* session will grow by leaps and bounds unless you take care to release the memory (by using `Clear`) used by lists that you don't need anymore.[3]

3.5.1 Operations on a single list

You can add expressions to a list at its beginning, end, or anywhere in between.

3. In version 3.0, the global variable `$HistoryLength` can be used to limit the number of inputs and outputs that *Mathematica* keeps track of in a session.

```
list3 = {a, b, c, d, e};

Append[list3, 42]
{a, b, c, d, e, 42}

Prepend[%, {x, y}]
{{x, y}, a, b, c, d, e, 42}

Insert[%, N[Pi], 4]
{{x, y}, a, b, 3.14159, c, d, e, 42}
```

Note that the original list has not been modified.

```
list3
{a, b, c, d, e}
```

The functions AppendTo and PrependTo work just as their counterparts Append and Prepend do, except that they modify their first argument. In other words, AppendTo[*list*, *elem*] is equivalent to *list* = Append[*list*, *elem*].

Make a copy of list3.

```
list4 = list3
{a, b, c, d, e}
```

The return value of AppendTo is the same as that of Append.

```
AppendTo[list4, 42]
{a, b, c, d, e, 42}
```

However, the original list has been modified.

```
list4
{a, b, c, d, e, 42}
```

You can replace any part of a list by another expression using ReplacePart:

```
ReplacePart[list3, new, 2]
{a, new, c, d, e}
```

ReplacePart does not modify the original list.

```
list3
{a, b, c, d, e}
```

You can specify a multidimensional index inside list braces.

```
ReplacePart[mat, new, {2, 3}]
{{a11, a12, a13}, {a21, a22, new}}
```

You can specify several independent indices using a nested list.

```
ReplacePart[mat, new, {{1}, {2, 3}}]
{new, {a21, a22, new}}
```

Starting with version 3.0, ReplacePart has been enhanced so that you also can specify which part of the new expression to use as a replacement. For example,

```
ReplacePart[{{a1, a2, a3}, {b1, b2, b3}},
            {{c1, c2}, {d1, d2}}, {{1}, {2, 2}}, {1, 2}]

{c2, {b1, c2, b3}}
```

There are more advanced uses of this technique that we will discuss in Section 7.3.2.

To extract contiguous items from a list, use `Take` or `Drop`. There are three forms of each function.

This takes the first 2 parts from the list.

```
Take[list3, 2]
{a, b}
```

This takes the last 2 parts from the list.

```
Take[list3, -2]
{d, e}
```

This takes parts 2 through 4 from the list.

```
Take[list3, {2, 4}]
{b, c, d}
```

`Drop` is used the same way as `Take`, but it extracts the complement of what `Take` does.

```
{Drop[list3, 2], Drop[list3, -2], Drop[list3, {2, 4}]}
{{c, d, e}, {a, b, c}, {a, e}}
```

You can rearrange the parts of a list by rotating, reversing, or sorting the list.

You can rotate in either direction.

```
{RotateLeft[list3, 1], RotateRight[list3, 2]}
{{b, c, d, e, a}, {d, e, a, b, c}}
```

```
Reverse[list3]
{e, d, c, b, a}
```

Here is a list of five random integers between 1 and 100.

```
Table[Random[Integer, {1, 100}], {5}]
{83, 77, 90, 9, 60}
```

```
Sort[%]
{9, 60, 77, 83, 90}
```

You can *flatten* a list, which removes extraneous braces. This is useful when built-in functions such as `Solve` return an unnecessarily deeply nested result.

```
Flatten[{{{a} , b}, {c, {d, {e, f}}}}]
{a, b, c, d, e, f}
```

You can restrict the amount of flattening with an optional second argument.

```
Flatten[{{{a}, b}, {c, {d, {e, f}}}}, 1]
{{a}, b, c, {d, {e, f}}}
```

```
Flatten[{{{a}, b}, {c, {d, {e, f}}}}, 2]
{a, b, c, d, {e, f}}
```

3.5.2 Operations on multiple lists

You can concatenate one list onto another.

```
Join[{a, b, c}, {d, e, f}]
{a, b, c, d, e, f}
```

Lists can also be treated as sets in the mathematical sense using the functions `Union`, `Intersection`, and `Complement`. Note that all of these functions return a list that is sorted and has no duplicates.

```
Union[{b, c, a, d}, {e, b, d}]
{a, b, c, d, e}

Intersection[{b, c, a, d}, {e, b, d}]
{b, d}

Complement[{b, c, a, d}, {e, b, d}]
{a, c}
```

Incidentally, calling any of these functions with a single list argument sorts the list and removes duplicate elements from it.

Exercises

1. Use `Join` to write functions that behave like `Append` and `Prepend`.
2. Write a function called `SubsetQ` that determines if its first argument is a subset of its second argument and returns `True` or `False`.

3.5.3 Mapping functions onto lists

Sometimes you want to apply a function to each part of a list. If it is a built-in, listable function, this is no problem; for example,

```
OddQ[{1, 2, 3}]
{True, False, True}
```

However, if the function is user-defined, this simple approach won't work. For example, suppose you would like to test each expression in a list to see if it is greater than 2.

First define an appropriate predicate.

```
ClearAll[greater2]
greater2[x_] := x > 2
```

Passing the entire list to the predicate doesn't work.

```
greater2[{1, 2, 3}]
{1, 2, 3} > 2
```

One way to create the desired result would be to give `greater2` the `Listable` attribute, as discussed in Section 3.3. This isn't always desirable, however, as we'll see below. Another strategy would be to use `Table` to construct each individual result:

```
Table[{1, 2, 3}[[i]] > 2, {i, 3}]
{False, False, True}
```

This solution is inelegant and inefficient, however. This kind of operation is so common that there is a built-in function, `Map`, that does it. `Map` simply wraps a given function around each part of a list.[4]

```
Map[f, {a, b, c}]
{f[a], f[b], f[c]}
```

In the current example, the function we want to map is `greater2`:

```
Map[greater2, {1, 2, 3}]
{False, False, True}
```

There also are many built-in functions that aren't listable. There's a simple reason for this: Any function that expects a list as an argument can't be listable, or it wouldn't work correctly. For example, consider the `First` function, which returns the first part of a list. If `First` were listable, then an expression like `First[{1, 2, 3}]` (which should return `1`) would be converted to `{First[1], First[2], First[3]}`, which is nonsense.

Why might you *want* to map a function like `First` onto a list? If the list had sublists, of course! Since a matrix is stored as a list of lists, in which each sublist is a row of the matrix, then extracting the first part from each sublist (row) is equivalent to extracting the first *column* of the matrix:

Here is a 2 × 2 matrix.

```
Array[a, {2, 2}]
{{a[1, 1], a[1, 2]}, {a[2, 1], a[2, 2]}}
```

And here is its first row.

```
First[%]
{a[1, 1], a[1, 2]}
```

On the other hand, here is its first *column*.

```
Map[First, %%]
{a[1, 1], a[2, 1]}
```

Because `Map` is so useful, there is a special input form for it that is fairly common.

```
greater2 /@ {1, 2, 3}
{False, False, True}
```

Exercises

1. Use `Map` and the built-in function `Last` to extract the last column of a matrix.
2. Use `Map` to extract the *second* column of a matrix. *Hint:* First define a function that takes a list as an argument and returns the second part of the list.

---

4. Note that this is not quite the same thing as making the function `Listable`, because it works only for functions of a single argument. A more general operation, `MapThread`, is discussed in Section 5.2.1.

3.5.4 Modifying lists in place

All of the operations we have seen so far (with the exception of `AppendTo` and `PrependTo`) are *side effect-free*. That is, they do not change the original list; instead, they make a copy of it and modify the copy, which is returned. When you are working with a very large list, however, it may save a lot of memory to modify the original list. Indexing can be used on the left side of an assignment operator for this purpose.

Note that the assignment operator returns the value being assigned, not the new list.

```
list5 = {1, 2, Sqrt[3]}
list5[[2]] = Log[7]
{1, 2, Sqrt[3]}

Log[7]
```

The original list has been modified.

```
list5
{1, Log[7], Sqrt[3]}
```

Working with large lists in this way can lead to some unexpected performance penalties. We'll deal with this problem in Section 10.2, "Procedural Perils."

3.6 Character Strings

Although character strings are not lists, we discuss them in this chapter because most of the operations on them are so similar to the operations defined for lists. Furthermore, it is a simple matter to convert a character string into a list of individual characters (and back again), so that any list operation can be applied to them.

3.6.1 Getting information about strings

`StringLength` and `StringPosition` are analogous to the `Length` and `Position` operations on lists.

Here's a string consisting of the first 10 letters of the alphabet.

```
s = "abcdefghij"
abcdefghij
```

`StringLength` is analogous to `Length`.

```
StringLength[s]
10
```

You can find all the positions of a given substring within a string. The return value is a list of `{start, end}` pairs.

```
StringPosition[s, "cd"]
{{3, 4}}

StringPosition["hello there", "he"]
{{1, 2}, {8, 9}}
```

`StringPosition` takes an option, `IgnoreCase`, that defaults to `False`.

```
StringPosition["Ada", "a"]
{{3, 3}}

StringPosition["Ada", "a", IgnoreCase->True]
{{1, 1}, {3, 3}}
```

Several predicates are defined for strings.

`s` is a string.

```
StringQ[s]
True
```

All characters in `s` are letters.

```
LetterQ[s]
True
```

All characters in this string are digits.

```
DigitQ["1234"]
True
```

These two should be self-explanatory.

```
{LowerCaseQ[s], UpperCaseQ[s]}
{True, False}
```

Finally, you can test a string to see if it matches a *string pattern*, which is another string that optionally contains wildcard characters.

```
StringMatchQ["the rain in spain", "*rain*"]
True
```

3.6.2 Operations on strings

As with lists, you can take or drop a given number of characters from the beginning, middle, or end of a string.

```
StringTake[s, 3]
abc

StringTake[s,-3]
hij

StringTake[s, {4, 7}]
defg

StringDrop[s, {4, 7}]
abchij
```

You can also insert a string at an arbitrary position within another string.

```
StringInsert[s, "012345", 3]
ab012345cdefghij
```

As with lists, negative indices count backward from the end.

```
StringInsert[s, "012345", -2]
abcdefghi012345j
```

You can join strings to produce a larger string. (This also works on a list of strings.)

```
StringJoin["Four", " ", "score", " and 7 years ago"]
Four score and 7 years ago
```

A handy infix notation for `StringJoin` is `<>`.

```
"Four" <> " " <> "score" <> " and 7 years ago"
Four score and 7 years ago
```

You can reverse a string.

```
StringReverse[s]
jihgfedcba
```

One list operation that has no counterpart for strings is `Sort` — presumably, the order of the characters in a string is significant, and you shouldn't want to sort them.[5]

The analogy with lists breaks down slightly when we come to the replacement function. Whereas in a list you replace parts based on their position (e.g., `s[[n]] = ...`), `StringReplace` replaces based on *content*. The replacements are specified as rules of the form *`oldstr->newstr`*.

```
StringReplace[s, {"cd"->"X", "h"->"YY"}]
abXefgYYij
```

`StringReplace` also accepts the `IgnoreCase` option demonstrated in the previous section.

The following functions have no analogues with respect to lists.

```
ToUpperCase[s]
ABCDEFGHIJ

ToLowerCase[%]
abcdefghij
```

Exercises

1. Implement a `StringCount` function that behaves as shown:

```
StringCount["abacab", "a"]
3

StringCount["abacab", "ab"]
2
```

2. Write a function that takes a string and constructs from it a *palindrome* (a string that reads the same in both directions).

5. Although you could easily write a function to do so (see the exercises in the next section).

3.6.3 Converting strings to/from other forms

You can break a string down into a list of characters.

```
Characters[s]
{a, b, c, d, e, f, g, h, i, j}
```

Note that each character is actually a string of length 1.

```
InputForm[%]
{"a", "b", "c", "d", "e", "f", "g", "h", "i", "j"}
```

Thus, `StringJoin` reverses this process.

```
StringJoin[%]
abcdefghij
```

Alternatively, you can convert a string to a list of ASCII codes using `ToCharacterCode`, and you can go in the reverse direction using `FromCharacterCode`.

```
ToCharacterCode[s]
{97, 98, 99, 100, 101, 102, 103, 104, 105, 106}

FromCharacterCode[%]
abcdefghij
```

You can convert *Mathematica* expressions to strings, and vice versa.

```
ToString[a + b + c]
a + b + c
```

Is that really a string?

```
InputForm[%]
"a + b + c"
```

This one is a little unusual.

```
ToString[a/b]
a
-
b
```

`\n` represents a *newline* character. This is unsuitable for a linear representation.

```
InputForm[%]
"a\n-\nb"
```

You can specify an output format when converting to a string.

```
ToString[InputForm[a/b]]
a/b
```

For more information about output formats, see Section 12.3.2, "Writing to a stream," or refer to [Wolfram 91] §2.7 or [Wolfram 96] §2.8.

Convert strings to *Mathematica* expressions using `ToExpression`.

```
ToExpression["a + b"]
a + b
```

```
FullForm[%]
Plus[a, b]
```

`ToExpression` will fail if the string is not syntactically valid *Mathematica* input.

```
ToExpression["2 +"]
ToExpression::sntxi:
   Incomplete expression; more input is needed.
$Failed
```

To avoid error messages, you can check whether or not a string is syntactically valid before attempting to do the conversion.

```
{SyntaxQ["2 +"], SyntaxQ["2 + 3"]}
{False, True}
```

When you convert a string to an expression, *Mathematica* attempts to evaluate it immediately.

```
ToExpression["2 + 3"]
5
```

You can prevent this behavior using `ToHeldExpression`.

```
ToHeldExpression["2 + 3"]
Hold[2 + 3]
```

The held expression can be evaluated at a later time.

```
ReleaseHold[%]
5
```

Exercises

1. Write a function that sorts the characters in a string.
2. Write functions to encrypt and decrypt a string by adding a constant to the character code for each character in the string.

3.7 Appendix: Listable Functions

Table 3-1 shows all of the functions with the `Listable` attribute in version 3.0 of *Mathematica*. Functions marked with an asterisk (*) do not exist in earlier versions; functions marked with a dagger (†) exist but do not have the `Listable` attribute in earlier versions. Functions with extremely long names appear at the end of the table.

Table 3-1 Listable functions in version 3.0.

Abs	AiryAi	AiryAiPrime	AiryBi
AiryBiPrime	Apart†	ArcCos	ArcCosh
ArcCot	ArcCoth	ArcCsc	ArcCsch
ArcSec	ArcSech	ArcSin	ArcSinh
ArcTan	ArcTanh	Arg	Attributes
BesselI	BesselJ	BesselK	BesselY
Beta	BetaRegularized	Binomial	Cancel

Table 3-1 (*Continued*) Listable functions in version 3.0.

Ceiling	Characters	ChebyshevT	ChebyshevU
Coefficient†	Conjugate	Cos	Cosh
CoshIntegral	CosIntegral	Cot	Coth
Csc	Csch	Denominator†	Divide
Divisors	DivisorSigma	EllipticE	EllipticF
EllipticK	EllipticPi	EllipticTheta	Erf
Erfc	Erfi	EulerPhi	EvenQ
Exp	ExpIntegralE	ExpIntegralEi	Exponent
Factor†	Factorial	Factorial2	FactorInteger
FactorSquareFree†	Floor	FractionalPart†	FresnelC
FresnelS	Gamma	GammaRegularized	GCD
GegenbauerC	HermiteH	Hypergeometric0F1	Hypergeometric1F1
Hypergeometric2F1	HypergeometricU	Im	In
InString	IntegerDigits	IntegerPart*	IntervalMemberQ†
JacobiP	JacobiSymbol	JacobiZeta	LaguerreL
LCM	LegendreP	LegendreQ	LerchPhi
Limit	Log	LogGamma	LogIntegral
MantissaExponent	MathieuC*	MathieuCPrime*	MathieuS*
MathieuSPrime*	MessageList	Minus	Mod
MoebiusMu†	Multinomial†	Negative	NonNegative
NonPositive*	Numerator†	OddQ	Out
PartitionsP†	PartitionsQ†	Plus	Pochhammer
PolyGamma	PolyLog	PolynomialGCD	PolynomialLCM
Positive	Power	PowerMod	Prime
PrimePi†	PrimeQ	ProductLog*	Quotient
Range	Re	RealDigits	Resultant
RiemannSiegelZ	Round	Sec	Sech
SetAccuracy	SetPrecision	Sign	Sin
Sinh	SinhIntegral	SinIntegral	Sqrt
StirlingS1†	StirlingS2†	Subtract	Tan
Tanh	Times	ToExpression	Together
ToHeldExpression	ToLowerCase†	ToUpperCase†	TrigFactor*
Zeta	$NumberBits		
ArithmeticGeometricMean		EllipticThetaPrime	
Hypergeometric0F1Regularized		Hypergeometric1F1Regularized	
Hypergeometric2F1Regularized		MathieuCharacteristicA*	
MathieuCharacteristicB*		MathieuCharacteristicExponent*	
RiemannSiegelTheta		SphericalHarmonicY	

Part 2
Programming Techniques

4

Procedural Programming

In this chapter we present those features of *Mathematica* that correspond most closely to the features of *procedural* programming languages such as C, C++, Fortran, and Pascal. Procedural programming is a style of programming that is characterized by lots of small steps and pervasive use of side effects (e.g., assignments to variables) to convey information between those steps. It also relies on iteration (e.g., loops) to process collections of data. Generally speaking, if you see assignment statements and loops, the style is procedural.

While most *Mathematica cognoscenti* discourage the use of procedural programming, it can be a useful stepping-stone into *Mathematica* programming for programmers whose experience lies mainly with the languages mentioned above. Bear in mind that most of the techniques illustrated in this chapter are but "tips of icebergs"; that is, they are simple manifestations of very powerful concepts that we will explore in greater detail in the chapters to come.

4.1 Functions

A *Mathematica function* is a programming construct that allows you to reuse pieces of code over and over in a convenient manner. Hundreds of functions, such as `Integrate`, `Table`, and `Plot`, are built into *Mathematica*. But *Mathematica* also makes it easy for you to define your own functions, which can be just as sophisticated as the functions provided by the system.

4.1.1 Function definition

Here is the definition of a function that computes the area of a circle of radius `r`:

```
area[] := Pi r^2
```

The expression on the left-hand side of the assignment is called the *declaration* of the function, and the expression on the right-hand side is called the *body* of the function. Whenever you use the function in an expression, the function is said to be *called*, and the value computed by the function (the *return value*) is substituted for the function call.

```
area[]
```

$$\text{Pi } r^2$$

The square brackets are important; without them the function won't be called.

```
area
```

```
area
```

You can see the definition of a function by asking for help on the name of the function, just as you would for a built-in function.

This shows the definition we have made for the symbol `area`.

```
?area
```

```
Global`area

area[] := Pi*r^2
```

Note, however, that the function name will not appear in the front end's *function browser* unless you put the function in a *package*. We'll discuss how you can create your own packages in Chapter 8, "Writing Packages."

4.1.2 Parameters

The expression returned by the `area` function involves a symbol called `r`. If this symbol has a value assigned to it, that value will be substituted into the body of the function when the function is called.

```
r = 2;
area[]
```

```
4 Pi
```

This is a cumbersome and error-prone way of getting values into a function; a better way is to use the mechanism of *parameters*.

Here is a redefinition[1] of `area` that uses a parameter instead of a global symbol.

```
Clear[area]
area[r_] := Pi r^2
```

The symbol `r` is the parameter. The underscore (`_`) is called `Blank`, which is intended to convey the idea "fill in the blank." When you call this function, for example, by evaluating

1. For reasons that will become much clearer (no pun intended) later, it's good practice to always `Clear` a function before making changes to its definition.

```
area[3]
9 Pi
```

the value 3 is substituted for `r` everywhere that `r` occurs in the body of the function. In computer science lingo, `r` is called the *formal parameter*, and the value 3 is called the *actual parameter* or *argument*. Note that the value returned by `area` no longer depends on the value of the global symbol `r` (which was set to 2 above). The global symbol `r` and the parameter `r` are two completely different entities.

The blank following the formal parameter in the declaration of a function is very important — it indicates that the parameter is indeed a formal parameter rather than a *literal value*.

If the underscore had been omitted, as in ...

```
Clear[area, r]
area[r] := Pi r^2
```

then `area` would evaluate only when the actual parameter is literally `r`,

```
area[r]
Pi r^2
```

but not for any other actual parameter.

```
area[2]
area[2]
```

Note that when the way a function is called does not match its definition, *nothing* happens! *A very common source of errors is to forget the blanks!* These errors can be notoriously hard to track down, since you don't get any kind of warning message, just a function call that "does nothing."

Here's a slightly more interesting example: a function with three parameters.

This function returns one root of the quadratic equation $ax^2 + bx + c = 0$.

```
quad[a_, b_, c_] := (-b + Sqrt[b^2 - 4 a c]) / (2 a)
```

One nice thing about symbolic computation is that it's very easy to see if a function returns the intended result: simply call it with symbolic arguments. However, you should try to get in the habit of using arguments whose names are different than the names of the parameters, for the reason explained above.

This is a poor test of the `quad` function.

```
quad[a, b, c]
```

$$\frac{-b + \text{Sqrt}[b^2 - 4\ a\ c]}{2\ a}$$

This is a much better test; it demonstrates that parameter substitution really is taking place.

```
quad[d, e, f]
```

$$\frac{-e + \text{Sqrt}[e^2 - 4\ d\ f]}{2\ d}$$

Here we substitute the result of `quad` into a quadratic equation.

```
d x^2 + e x + f /. x->% // Expand
0
```

Exercises

1. The *law of cosines* states that if a, b, and c are the sides of a triangle, then $a^2 = b^2 + c^2 - 2ab\cos\theta$, where θ is the angle between sides b and c. Write a function that returns the length of a when given b, c, and θ.
2. Write a function that takes a list as a parameter and returns the middle element of the list.
3. Write a function that takes as parameters a function of some independent variable and the independent variable itself, and returns a list of rules giving all critical points of that function (values of the independent variable where the first derivative is 0), e.g.,

```
critpts[x^2 - 1, x]
{{x -> 0}}

critpts[y^3 + 10 y^2 + 5, y]
{{y -> -(20/3)}, {y -> 0}}
```

4. Write a function that takes three numerical parameters and returns `True` if they are in sorted order, `False` otherwise.

4.1.3 Type checking

Most procedural programming languages allow (or require!) a programmer to specify that the arguments to a function must be of a certain *type* such as integer or real. In *Mathematica*, every expression has a *head* that can be considered to be its "type."

The head of any expression can be found using `Head[expr]`.

```
Map[Head, {3, 3/2, 3 + 2 I, {3, 2}, a}]
{Integer, Rational, Complex, List, Symbol}
```

It is possible to specify that an argument to a function must have a certain head by using a formal parameter of the form *`name_head`* in the definition of the function. For example, the factorial function is defined only for integer arguments, so we might write it like this:

```
fact[n_Integer] := Product[i, {i, n}]
```

`fact` will evaluate for integer arguments...

```
fact[4]
24
```

but not for any other type.

```
fact[3.5]
fact[3.5]
```

Similarly, a function requiring a list as an argument could be defined as *`func`*`[s_List]`.

The kind of type checking afforded by the construct *name_head* is syntactic, rather than semantic, in nature. In other words, even though every integer is a real number and every real number is also complex, you can't pass an integer to a function that is declared to take a real argument, or a real number to a function that is declared to take a complex argument. There is no automatic *type promotion* of arguments as there is in some other languages. This should be considered advantageous, as it gives the programmer tighter control over how a function can be called. It is still possible to write a type-checked function that takes more than one type of argument. One way to do this is shown below:

The vertical bar operator is called `Alternatives`.

The function operates on any type of numerical argument.

```
squareAnyNumber[n_Integer | n_Real |
                n_Rational | n_Complex] := n^2

Map[squareAnyNumber, {3, 4.5, 2/3, 3 + 4 I, x}]

{9, 20.25, 4/9, -7 + 24 I, squareAnyNumber[x]}
```

Furthermore, you can use `Head` to test the type of an argument, and react accordingly. Here is an example of a factorial function that is extended to the entire complex plane.[2]

The `If` function will be covered in detail in Section 4.2.1.

```
extendedfactorial[n_Integer | n_Real |
                  n_Rational | n_Complex] :=
    If[Head[n] == Integer,
        (*then*) Product[i, {i, n}],
        (*else evaluate a power series for Gamma[n+1]*)
    ]
```

A more elegant way to accomplish the same thing is to create multiple definitions of the `extendedfactorial` function, each one taking different types of arguments:

```
Clear[extendedfactorial];
extendedfactorial[n_Integer] := Product[i, {i, n}]
extendedfactorial[n_Real | n_Rational | n_Complex] :=
    (* power series for Gamma[n+1] *)
```

When the argument to `extendedfactorial` has head `Integer`, the first definition will be used; when the head is `Real`, `Rational`, or `Complex`, the second definition will be used. This is analogous to *function-name overloading* in languages such as C++ [Stroustrup 91]. In *Mathematica*, however, it is a simple use of a much more general mechanism, *pattern matching*, that we will discuss in Chapter 6.

Exercise

1. Define a function of two integers, `n` and `r`, that computes the binomial coefficient `n!/(r!(n - r)!)`.

2. The built-in factorial function is extended in this sense.

4.1.4 Local variables

Suppose we want to modify the quad function introduced in Section 4.1.2 to return a list containing *both* roots of the quadratic equation. Since the value of the radical is used twice (once for each root), it would be sensible to compute it only once and store it in a variable. Here is how that could be done:

```
Clear[quad]
quad[a_, b_, c_] := (
    d = Sqrt[b^2 - 4 a c];
    {-b + d, -b - d} / (2 a)
)
```

Here are both roots of `x^2 - 3 x + 2 == 0`.

```
quad[1, -3, 2]
{2, 1}
```

The quad function now consists of a `CompoundExpression` having two subexpressions (Section 2.3.7); these subexpressions correspond to *statements* in procedural programming languages. The entire `CompoundExpression` must be surrounded by parentheses, or else the parser would assume that the function ended after the first semicolon it encountered. Note that there is no semicolon following the final subexpression in the function — if there were, the function would return `Null`.

There's a potentially serious problem with this definition of quad: It produces a side effect in the global name space. In other words, if we already had defined a symbol named d, its value would be trashed whenever quad was called!

d has been overwritten by the value of the discriminant.

```
d = 0;
quad[1, 1, 1]
d
```

$$\left\{\frac{-1 + I\ \text{Sqrt}[3]}{2},\ \frac{-1 - I\ \text{Sqrt}[3]}{2}\right\}$$

```
I Sqrt[3]
```

To avoid this problem, a symbol like d in the quad function should be made *local* to that function. This can be done by enclosing the body of the function in a `Module`. The syntax of a `Module` is `Module[`*`declarations, body`*`]`, where *`declarations`* contains a list of the symbols in *`body`* that are to be considered local.

Note that parentheses are no longer necessary because the square brackets of the `Module` serve to delimit the body of the function.

```
Clear[quad]
quad[a_, b_, c_] :=
    Module[{d},
        d = Sqrt[b^2 - 4 a c];
        { -b + d, -b - d } / (2 a)
    ]
```

The value of the global symbol d no longer is affected by the call to quad.

```
d = 0;
quad[1, 1, 1]
d
```

$$\left\{\frac{-1 + I\ Sqrt[3]}{2}, \frac{-1 - I\ Sqrt[3]}{2}\right\}$$

```
0
```

The *declarations* argument to Module allows you to specify multiple local symbols and, optionally, initial values for any of those symbols.

This Module declares two local symbols and initializes them.

```
Clear[quad]
quad[a_, b_, c_] :=
    Module[{d = Sqrt[b^2 - 4 a c], e = 2 a},
        {-b + d, -b - d} / e
    ]

quad[1, 0, -16]
{4, -4}
```

However, initial values should depend only on global symbols and parameters to the function, not on other local symbols.

An alternative to Module, for use in cases in which you need to assign a value to a local symbol only once, is the With function. With declares local *constants*, rather than local variables; it is similar to the let function of Lisp. The example shown above could have been written this way:

```
quad[a_, b_, c_] :=
    With[{d = Sqrt[b^2 - 4 a c], e = 2 a},
        {-b + d, -b - d} / e
    ]
```

Note that you cannot assign to d or e inside the body of the With. This is because the mechanism used by With is fundamentally different from that of Module: You can think of With[{*name* = *expr*}, *body*] as textually substituting the value of the expression *expr* everywhere for *name* in *body*. There really is no symbol named *name*, so no assignment is possible.

On the other hand, because With doesn't create any new symbols, it is slightly faster than Module. Furthermore, it is good coding style to use With instead of Module in cases where With will do, because it alerts someone who is trying to understand your code that the symbols declared therein are constants. It should come as no surprise that these arguments are exactly the same as the ones for using #define declarations in C or const declarations in C++.

In order to make some local symbols variables and others constants, it is necessary to nest a Module within a With or vice versa.

Exercise

1. The curvature $K(t)$ of the parametric arc $s(t) = (x(t), y(t), z(t))$ is defined as

$$K(t) = \left|\frac{T'(t)}{|v(t)|}\right|$$

where $v(t) = s'(t)$ and $T(t) = v(t)/|v(t)|$. Write a function that takes a parametric arc (in the form of a list of three functions) and an independent variable and returns $K(t)$. Make the velocity and its magnitude local symbols. You can check your answer by showing that the curvature of the circular helix $(\cos t, \sin t, t)$ is 1/2 (independent of t).

4.1.5 Set versus SetDelayed

In the examples given so far we have used `SetDelayed` (`:=`) to define functions. The difference between `SetDelayed` and `Set` (`=`) is that the right-hand side (RHS) of a `Set` is evaluated immediately, whereas the RHS of a `SetDelayed` is not. `SetDelayed` is almost always what you want to use to define a function, and here's why:

Using `Set`, the current value of `a` (3) is substituted into the RHS of the function definition.

```
a = 3;
foo[a_] = a^3
27
```

The function always returns 27.

```
foo[2]
27
```

This is another source of hard-to-find bugs. Fortunately, checking the definition of your function by using `?name` shows this type of error immediately.

```
?foo
Global`foo

foo[a_] = 27
```

However, there are occasions on which `Set` is exactly what you need. Suppose you have some expression resulting from another calculation, and you want to create a function that returns that expression. The following will not work:

The intention here is for `g[x]` to evaluate to the first three terms in the Taylor series approximation for `BesselJ[0, x]`.

```
Normal[Series[BesselJ[0, x], {x, 0, 3}]]
```

$$1 - \frac{x^2}{4} + \frac{x^4}{64}$$

```
g[x_] := %
```

What happened?

```
(* ... more calculation... *)
Log[17.5];
g[z]
2.8622
```

Looking at the definition of g provides a clue:

```
?g
Global`g

g[x_] := %
```

The literal % operator appears in the definition of g. In other words, every time g is called, you are reevaluating whatever result immediately preceded the call — in this case, `Log[17.5]`.

The correct thing to do in this case is to use `Set`, so that the RHS is evaluated at the point of function definition.

The value returned is the RHS of the function definition — i.e., the % has been expanded.

```
Clear[g]
Normal[Series[BesselJ[0, x], {x, 0, 3}]];
g[x_] = %
```

$$1 - \frac{x^2}{4} + \frac{x^4}{64}$$

Now g works as expected.

```
Log[17.5];
g[z]
```

$$1 - \frac{z^2}{4} + \frac{z^4}{64}$$

Another case in which you might want to use `Set` for function definition is when the RHS can be simplified symbolically, even before the substitution of any parameters. If the simplification saves a significant amount of computation, you probably want it to be done in advance. For example, the following symbolic integration is fairly slow:

The error message occurs only in version 3.0.

```
f[a_, b_] := Integrate[BesselJ[0, x], {x, a, b}]

Timing[Table[f[0, c], {c, 1, 3}];]
Integrate::gener: Unable to check convergence
{4.51667 Second, Null}
```

By using `Set`, the integral is evaluated before the assignment to `f`.

```
Clear[f]
f[a_, b_] = Integrate[BesselJ[0, x], {x, a, b}]
Integrate::gener: Unable to check convergence
```

$$-(a\ \text{HypergeometricPFQ}[\{\tfrac{1}{2}\}, \{1, \tfrac{3}{2}\}, \tfrac{-a^2}{4}]) -$$

$$b\ \text{HypergeometricPFQ}[\{\tfrac{1}{2}\}, \{1, \tfrac{3}{2}\}, \tfrac{-b^2}{4}]$$

A large speedup is achieved.

```
Timing[Table[f[0, c], {c, 1, 3}];]
{0.183333 Second, Null}
```

In the second definition of `f` the integration was performed only once, when the function was defined. This results in a 10-fold speedup in evaluation, which would be really appreciable if `f` were to be evaluated many times, for example, when plotting.[3]

Exercises

1. Here is another common mistake:

```
Normal[Series[BesselJ[0, x], {x, 0, 3}]];
Clear[g]
g[x_] = %
```

   Execute this code and explain what went wrong.

2. Write a function that evaluates the derivative of the `BesselJ[0, x]` function for any `x` (including numerical values). Why is this easier to do using `Set` than using `SetDelayed`?

4.1.6 Return values

Functions should always return their values using the normal function return mechanism [i.e., the last expression evaluated by the function, or the argument to an explicit call to `Return` (Section 4.3.4), is the return value of the function].

It is considered the poorest of style to print results using `Print` rather than to return them, because printed results are a computational dead-end — they cannot be used as input to other commands. Even if a function returns highly formatted data, it is better to return the raw data (in a nested list, if necessary) and allow the caller to format it as desired. Better yet, write a second function that formats the data returned by the first function.

4.2 Conditional Execution

Sometimes your code needs to take different actions depending on the value of some expression. *Mathematica* provides three conditional execution primitives: `If`, `Switch`, and `Which`. Unlike procedural languages, however, these primitives are themselves function calls, rather than syntactic keywords.[4]

---

3. Of course, if the original intent were to plot the values of the integral, it would have been speedier still to define `f[a_, b_] := NIntegrate[BesselJ[0, x], {x, a, b}]`. In this case the use of `Set` would cause an error (why?).
4. This is a reflection of the fact that, underneath the procedural veneer, *Mathematica* is actually a functional programming language — the topic of the next chapter.

4.2.1 If

The `If` function takes three arguments: the *test*, the *if* branch, and the *else* branch.

This function calculates the absolute value of its argument.

```
absval[x_] := If[x > 0, x, -x]
Table[absval[i], {i, -3, 3}]
{3, 2, 1, 0, 1, 2, 3}
```

Each of the arguments can be a list of expressions separated by semicolons, so *Mathematica*'s `If` is just as powerful as any syntactic *if* statement. For example,

The `(*then*)` and `(*else*)` are merely comments. The statements between them constitute a single argument to the `If` function.

```
a = 5; b = 2;
If[a < b,
(*then*)
    little = a;
    big = b,
(*else*)
    little = b;
    big = a
]
5

{little, big}
{2, 5}
```

Note that `If` returns the result of the last expression evaluated within its body.[5] As always, this can be suppressed with a semicolon.

Since one or more of the arguments to a boolean operation could be symbolic, it is quite possible that the operation cannot be evaluated. If this happens, `If` will be unable to choose either alternative. For example,

```
Clear[a, b]
If[a < b, a, b]
If[a < b, a, b]
```

To handle cases such as this, `If` takes an optional fourth parameter:

```
If[a < b, a, b, "I give up"]
I give up
```

Sometimes you want to execute the *else* part even if the test expression can't be evaluated. Rather than using a four-parameter `If` with identical third and fourth parameters, wrap the first argument inside the `TrueQ` function.

---

5. Thus, the *Mathematica* expression `x = If[a, b, c]` is analogous to the C statement `x = a ? b : c`. Note, however, that the roles of semicolons and commas are reversed relative to C, e.g., *Mathematica*'s `x = If[a, b1; b2, c1; c2]` corresponds to C's `x = a ? b1, b2 : c1, c2`.

TrueQ returns true if its argument is *manifestly true*, and returns false otherwise.

```
If[TrueQ[a], "a is true", "a is not true"]
a is not true
```

Note that "a is not true" is not the same as "a is false." In this particular case, a is neither true nor false.

```
If[a, "a is true", "a is false", "a is neither"]
a is neither
```

Exercises

1. Modify the binomial function of Exercise 4.1.3.1 so that it returns 0 if either parameter is negative.
2. Write a function that takes three numbers a, b, and c and returns True if the three numbers could represent the lengths of the sides of a right triangle, False otherwise.
3. Write a function that takes three numerical parameters and returns true if they are in sorted order, false otherwise.

4.2.2 Switch

The Switch function is used to test the value of an expression against several different alternatives. It is similar to the C statement of the same name. The first argument to Switch is an expression to be tested; subsequent arguments, which must come in pairs, consist of alternative values for the tested expression and the actions to be taken for each possible value.

This example does one of three things, depending on whether the remainder of x after division by 3 is 0, 1, or 2.

```
x = 88;
Switch[Mod[x, 3],
       0, "remainder is 0",
       1, "remainder is 1",
       2, "remainder is 2"
]
remainder is 1
```

If the expression does not match any of the available cases, the Switch remains unevaluated.

```
Switch[Mod[x, 5],
       0, "remainder is 0",
       2, "remainder is 2",
       4, "remainder is 4"
]
Switch[Mod[x, 5], 0, remainder is 0, 2,
  remainder is 2, 4, remainder is 4]
```

A default case can be specified using a Blank (_).

```
Switch[Mod[x, 5],
       0, "remainder is 0",
       2, "remainder is 2",
       4, "remainder is 4",
```

```
        _, "remainder is odd"
]
remainder is odd
```

Exercise

1. Write a function called `daysInMonth[month, year]` that returns the number of days in the given month of the given year. (*Hint:* A default case will dramatically reduce the number of cases you have to program explicitly.) Don't forget about leap years!

4.2.3 Which

The `Which` statement is an alternative to a sequence of `if-elseif-elseif-...-else` statements. It is similar to the *cond* statement of Lisp.

```
Which[x < 0, "negative",
      x == 0, "zero",
      x > 0, "positive"]
positive
```

In this example, the three cases cover all possibilities. As with `Switch`, it sometimes is desirable to have a default case. Such a case can be implemented by specifying a final condition of `True`.

```
Which[x < 0, "negative",
      x == 0, "zero",
      True, "positive"]
positive
```

Exercise

1. `Which` is useful for defining piecewise functions. Write a function of a single variable `x` that is equal to `Sqrt[-x - 1]` when `x <= -1`, to `Sqrt[x - 1]` when `x >= 1`, and to `1 - x^2` when `x` is between `-1` and `1`. Plot this function from `x = -2` to 2.

4.3 Iteration

Mathematica contains three looping constructs that are found in other procedural programming languages: `Do`, `While`, and `For`. However, as with conditional execution, these primitives are all functions; there is no built-in syntax to support loops.

`Do`, `While`, and `For` all share an unusual trait (for *Mathematica* functions, that is): They don't return any values.[6] To get any useful results out of them requires the use of

6. To be pedantic, they return `Null`.

side effects — assignments to variables, `Print` statements, or graphics-generating commands such as `Plot`.

As we'll see throughout this book, there almost always are superior alternatives to the looping functions, but they require a different kind of programming style. We'll cover those other programming styles in the next two chapters.

4.3.1 Do

The `Do` function is a concise way to iterate over an index variable that takes on values in an arithmetic progression. It takes the same arguments as `Table` and `Sum`; unlike those functions, however, `Do` returns `Null`. Compare the following:

`Table` returns a list of the first argument evaluated at each value in the range of the index variable.

```
Table[k, {k, 3, 9, 2}]
{3, 5, 7, 9}
```

`Do` does all the work, but returns nothing; this was a waste of processor time!

```
Do[k, {k, 3, 9, 2}]
```

In order to do any useful work, the code inside of a `Do` must create some kind of side effect. Examples of side effects are: creating, removing, or changing the values of symbols; output to a file; or the rendering of graphics on the screen.

Here's one example of using side effects to get results out of a `Do` loop.

```
s = {};
Do[AppendTo[s, k], {k, 3, 9, 2}]
```

To result has been stored in `s`.

```
s
{3, 5, 7, 9}
```

`Sum` returns the sum of the first argument evaluated at each value in the range of the index variable.

```
Sum[k, {k, 3, 9, 2}]
24
```

Here is how to get the equivalent effect with `Do`.

```
s = 0;
Do[s += k, {k, 3, 9, 2}]
s
24
```

Here is a particularly useful trick for generating a sequence of animation frames. The output is not shown because it would take up several pages.

```
Do[Plot[Sin[x - y], {x, 0, 2Pi}], {y, 0, 15Pi/8, Pi/8}]
(* many graphics omitted *)
```

Like the `Table` command, `Do` can accept multiple index variables and ranges. This feature allows you to write nested loops in a very concise manner:

Note that the first index varies most slowly.

```
Do[Print[{x, y}], {x, 2}, {y, 3}]
{1, 1}
{1, 2}
{1, 3}
{2, 1}
{2, 2}
{2, 3}
```

The iteration variable in a `Do` loop (or `Table`, `Sum`, and similar commands) is rather unusual. Consider the following example carefully:

`foo` behaves as if the global symbol `k` were being modified inside the `Do`.

```
Clear[k, foo]
foo := k + 1
Do[Print[foo], {k, 0, 1}]
1
2
```

Curiously, though, `k` is unchanged by the execution of the `Do` loop.

```
k
k
```

We'll explore this phenomenon in greater detail in Section 4.5.

Exercises

1. Write a function that performs matrix multiplication using a multivariable `Do` loop. You may use the `Sum` function to compute the individual elements of the result. Note that you have to create an empty result matrix (using `Table[0, {nrows}, {ncols}]`) before you can assign to its elements. Check your answer using the built-in matrix multiplication function `Dot`.
2. Now show how you can write the matrix multiplication function using only `Table` and `Sum`.

4.3.2 For

`For` implements a C-style *for* loop. Like its C counterpart, `For` takes four arguments: the initializer, the test, the increment, and the body. Here is how you would implement the summation example from the previous section using `For`:

Like `Do`, `For` returns `Null`; you have to evaluate `s` to get the result of the computation.

```
s = 0;
For[i = 3, i <= 9, i += 2, s += i];
s
24
```

`For` is more general than `Do` because the initializer, test, and increment can be arbitrary *Mathematica* expressions. For the same reason, a `For` loop is slower than an equivalent `Do` loop, not to mention harder to parse by the human eye, so `Do` should be given preference whenever circumstances permit. Also note that to implement an iteration over more than one variable, `For` commands must be nested explicitly (i.e., there is no multivariable form of `For` as there is for `Do`).

A common mistake for C programmers is to use semicolons to separate the arguments to `For`. Remember, function arguments are separated by commas; semicolons are used only to construct compound statements. In fact, *Mathematica*'s use of commas and semicolons within a `For` command's first three arguments is *exactly the opposite* of C's.

Exercises

1. Write a function that finds the smallest element in a list of numbers.
2. Write a function that finds the smallest element in a 2-D *matrix* of numbers.
3. Use your `daysInMonth` function (Exercise 4.2.2.1) to write a function that returns the *Julian date* of a {month, day, year} date. [The Julian date is the number of days elapsed since the beginning of the year. For example, January 1 has Julian date 1 and December 31 has Julian date 365 (366 in leap years).]
4. Implement the extended definition of the binomial coefficients, valid for all real numbers n and all integers r, as follows:

$$\binom{n}{r} = \begin{cases} \dfrac{n(n-1)\cdots(n-r+1)}{r(r-1)\cdots 1} & r \text{ a positive integer} \\ 1 & r = 0 \\ 0 & r \text{ a negative integer} \end{cases}$$

4.3.3 While

Another looping construct borrowed from C is `While`, which takes two arguments, a test and a body. Here is the same summation example, this time using `While`:

Since the `While` loop does not return a value, `s` must be returned explicitly.

```
i = 3; s = 0;
While[i <= 9, s += i; i += 2];
s
24
```

What appears at first to be a third argument to `While` (`i += 2`) actually is a continuation of the second argument: This is evidenced by the fact that the two expressions are separated by a semicolon, not a comma. This syntax may take a while (no pun intended) for C programmers to get used to.

It should be noted that `For` and `While` are equally powerful; i.e., any `While` loop can be rewritten as a `For` loop and vice versa. Which one to use is a matter of stylistic preference.

`While` loops typically are used to loop for an unknown number of times, such as during an iterative approximation. Here is a function that approximates the square root

of a number x using the iterative formula $a_{n+1} = \frac{1}{2}(a_n + x/a_n)$:

In a symbolic system like *Mathematica*, it is important to numerically evaluate all inputs to an iterative approximation procedure or else it may never converge!

```
squareroot[x_] :=
Module[{approx = nx = N[x]},
    While[ Abs[nx - approx^2] > 10^-5,
           approx = (approx + nx/approx)/2
    ];
    approx
]

squareroot[5]
2.23607

%^2
5.
```

Exercises

1. Write functions to compute $n!$ using (a) `For`, (b) `While`, and (c) `Product`. Test the speed of your functions against that of the built-in factorial function.
2. Write a function that converts an integer into a list of its digits, using the following procedure: (a) Find the quotient and remainder upon division by 10, (b) digit = remainder, (c) new number = quotient. Repeat this procedure until the quotient is 0. (When you are done, the digits will be in reverse order; you can use the `Reverse` function to fix this.)

   Now generalize this function to work for any number base $b \leq 10$. The base b should be an additional argument to the function.
3. A simple method for finding the root of a continuous function f is called *bisection*. This approach begins with an interval (a, b) for which $f(a)$ and $f(b)$ have opposite signs. (Since f is continuous, there is guaranteed to be *at least* one root within the interval.) The interval is bisected, say, at point c. Now if $f(a)$ and $f(c)$ have the same sign, change the interval to (c, b). Otherwise [$f(b)$ and $f(c)$ have the same sign], change the interval to (a, c). Repeat this process until the interval is sufficiently small.

   Write a function to perform the bisection method. Test your function on $f(x) = x^2 - 2$, using an initial interval (0, 2). *Hints:* The `Sign` function is a quick way to determine the sign of a number. Be sure you do real arithmetic (as opposed to symbolic), or it will never converge! Also, the `Chop` function is mighty handy when you are testing for convergence.

4.3.4 Miscellaneous control flow

There are functions called `Break`, `Continue`, `Return`, `Goto`, and `Label` that are identical to the C statements of the same name. `Continue[]` skips the remainder of the current iteration of a loop, whereas `Break[]` exits the loop entirely. `Return[val]` exits a

function and causes the function to return *val*. `Goto` and `Label` probably require no introduction; you should avoid using go-to statements in *any* computer language, as they make your code hard to understand. See *The Mathematica Book* §2.5.9 for some restrictions on the use of `Break`, `Continue`, and `Goto`.

`Abort[]` causes an entire computation to halt and return the value `$Aborted`. It is possible to intercept aborts using the `CheckAbort` function; refer to *The Mathematica Book* for the details. A more refined alternative to `Abort` and `CheckAbort` is `Throw` and `Catch`, which implements a form of *exception handling*. `Throw` and `Catch` will be discussed in Section 5.3.4.

Exercises

1. Write a function that takes an expression, `target`, and a list of expressions, `candidates`, and returns the position of the first occurrence of `target` within `candidates`. The function should return an impossible value, such as -1, if no match is found.
2. Write a function `allodd[x]` that takes a list as a parameter and returns `True` if and only if all of the elements in the list are odd integers. (The built-in function `OddQ` can be used to test a number for "oddness.") For example,

```
allodd[{1, 2, 3, 4}]
False

allodd[{1, 3, 5, 7}]
True
```

Your code should stop testing as soon as it finds a number that is not odd.

4.4 Parameter-Passing Semantics

4.4.1 Parameters are *not* local variables

It will come as a surprise to programmers who are used to other programming languages that *you cannot modify a parameter of a function inside the body of the function*. For example:

This function tries to use its parameter as a loop counter.

```
sumdown[n_] := Module[{s = 0},
    While[n > 0, s += n; n--];
    s
]
```

Not only do we get this strange error message, but also we have to abort the computation.

```
sumdown[3]
Decrement::rvalue: 3 has not been assigned a value.
$Aborted
```

The reason for this lies in the way in which *Mathematica* passes arguments to functions. The arguments to a function are *evaluated* at the point of call, and then the results of those evaluations are *textually substituted* for the formal parameters within the body of the function. It is as though we had entered the following:

It should be clear now why it was necessary to abort the computation.

```
While[3 > 0; s += 3; 3--]
Decrement::rvalue: 3 has not been assigned a value.
$Aborted
```

This mechanism may seem similar to a macro substitution, but in fact it is different because of the evaluation of the arguments before the substitution is performed. For example, if we pass as the argument to `sumdown` a symbol that has the value 3, the same error will occur.

`p` is evaluated before being passed into `sumdown`.

```
p = 3;
sumdown[p]
Decrement::rvalue: 3 has not been assigned a value.
$Aborted
```

Another way of looking at this is that the semantics of parameter passing are identical to the semantics of the `With` command introduced in Section 4.1.4. Recall that `With[{`*`name = expr`*`}, `*`body`*`]` evaluates *`expr`* and then substitutes that value for *`name`* everywhere within *`body`*.

4.4.2 Call by value

The solution to the problem demonstrated in the previous section is, of course, to use a local variable as a proxy for the parameter.

Although you can't modify `n`, you certainly can modify `x`.

```
Clear[sumdown]
sumdown[n_] :=
Module[{x = n, s = 0},
    While[x > 0, s += x; x--];
    s
]

sumdown[3]
6
```

This effectively simulates *call by value* parameter-passing semantics, which is the semantic model used for all parameters in C [Kernighan & Ritchie 78] and for non-*var* parameters in Pascal [Jensen & Wirth 74].

4.4.3 Call by name

Call by name is a fairly esoteric semantic model for parameter passing that was introduced by Algol-60 [Naur 63]. The semantics of call by name are that a function should

behave as though it were expanded inline at the point of call (e.g., as if it were a macro), with the actual parameters textually substituted for the formal parameters. Although call by name seems painfully straightforward, in fact it is fraught with subtleties and booby traps.

Mathematica contains a mechanism that provides call by name semantics, namely, *held arguments*. Recall that some of the built-in functions do not evaluate their arguments in the normal way (e.g., `Set` does not evaluate its first argument, and `SetDelayed` does not evaluate either of its arguments). You can prevent any of the arguments to a function from being evaluated by giving the function the attribute `HoldAll`.[7]

Here is an example of a function with a held argument.

```
SetAttributes[inc, HoldAll]
inc[x_] := x = x + 1
```

Initialize p to 3.

```
p = 3;
```

This call is equivalent to `p = p + 1`.

```
inc[p];
```

p is now 4.

```
p
4
```

Incidentally, `Clear` clears only the definition of a symbol, not its attributes. When you start fooling around with attributes, it is best to get in the habit of using `ClearAll` instead of `Clear`. `ClearAll` clears values, attributes, default values, and other information about a symbol.

The effect of call by name is that a parameter is *reevaluated every time it is used* within the body of the function. This can lead to problems that will be familiar to users of the C language's macro preprocessor:

This function is trivially correct. Or is it?

```
SetAttributes[double, HoldFirst]
double[x_] := x + x
```

No problems here ...

```
double[p]
8
```

But if you were expecting 10 from this call, guess again!

```
double[inc[p]]
11
```

The expression `inc[p]` was evaluated *twice* during execution of `double`. Behavior such as this probably is a bug. You can fix it by copying the parameter into a local variable at the beginning of the function; then it is evaluated just once.

---

7. Similarly, the attribute `HoldFirst` prevents just the first argument to a function from being evaluated, and the attribute `HoldRest` prevents all *but* the first argument from being evaluated.

One function's bug is another one's feature: *Jensen's device* [MacLennan 83] is a trick that exploits the behavior just demonstrated to implement iteration functions like `Sum`, `Product`, `Table`, and `Do`.

Because `mysum` has the `HoldAll` attribute, `summand` will be evaluated each time it is used.

```
SetAttributes[mysum, HoldAll]
mysum[summand_, variable_, low_, high_] :=
Module[{thesum = 0},
    For[variable = low, variable <= high, variable++,
        thesum += summand
    ];
    thesum
]
```

Here's an example. To understand why this works, perform a textual replacement of the parameters in the body of the function.

```
j = 0;
mysum[Sin[j], j, 1, 5]
Sin[1] + Sin[2] + Sin[3] + Sin[4] + Sin[5]
```

An unfortunate side effect is that `j` has been altered. This will be fixed in the exercises.

```
j
6
```

Call by name is powerful and dangerous. Legitimate uses do exist in a symbolic algebra system such as *Mathematica*: Sometimes you need to operate on a symbolic expression rather than on the value to which the expression would evaluate (the `mysum` function is an example). Quite often, however, there are alternatives that are safer. Unless you are writing functions that truly need this capability, you are better off staying away from it.

As a final warning to the skeptics, consider the infamous call by name `swap` function:

This function is designed to swap its arguments.

```
SetAttributes[swap, HoldAll]
swap[a_, b_] :=
Module[{temp},
    temp = a;
    a = b;
    b = temp;
]
```

It seems to work just fine.

```
{p, q} = {1, 2};
swap[p, q]
{p, q}
{2, 1}

p = 1; t = {3, 2, 4};
```

Since `p` is `1`, this swaps `t[[1]]` and `p`.

```
swap[t[[p]], p]
{p, t}
{3, {1, 2, 4}}
```

When you can figure out what this does, you will be ready to use call by name.

```
p = 1; t = {3, 2, 4};
swap[p, t[[p]]]
{p, t}
{3, {3, 2, 1}}
```

Before you try to "fix" `swap`, it seems only fair to inform you that it has been shown (see [Fleck 76]) that it is theoretically impossible to write a call by name swap function that works for all possible parameters.

Exercises

1. Explain why functions such as `Set` and `Clear` have to hold their first argument.
2. Fix `mysum` so that the global loop index variable is not modified by the execution of the function.
3. What happens if you call `mysum[Sin[j], j, 1, inc[p]]`? Fix this.
4. What happens if you call `swap[3, 4]`?
5. Write a function that takes the same arguments as `While` but mimics the semantics of the `do ... while` construct of C (or the `repeat ... until` construct of Pascal). That is, the body of the loop should always execute at least once. You will need to give the function the attribute `HoldAll` (why?).

4.4.4 Call by reference

Call by reference is a semantic model for parameter passing that allows changes made to a parameter inside of a function to be visible outside the function. This is the model used by all Fortran parameters, by C++ *reference* parameters [Stroustrup 91], and by Pascal *var* parameters. It typically is used by functions that need to return more than a single value, although it is preferable in such cases to return a list of values.

Mathematica's call by name semantics can be used to simulate call by reference behavior in a "safe" fashion. The trick [Maeder 94a] is to give the function one of the `Hold-` attributes *and* to specify that the arguments to be modified be symbols rather than arbitrary expressions. The latter can be effected by using a formal parameter of the form *name*`_Symbol`. For example:

Since you cannot pass arbitrary expressions to `newswap`, it is safer than `swap`.

```
SetAttributes[newswap, HoldAll]

newswap[a_Symbol, b_Symbol] :=
Module[{temp},
    temp = a;
    a = b;
    b = temp;
]

{p, q} = {1, 2};
newswap[p, q]
{p, q}
{2, 1}
```

Note that this isn't really call by reference, because there is no such thing as a *pointer* in *Mathematica*. If, for example, you pass a subscripted list to `newswap`, nothing will happen because the list subscripting expression doesn't have the head `Symbol`:

```
newswap[p, t[[p]]]
newswap[p, t[[p]]]
```

On the bright side, this same feature prevents errors when calls such as this are attempted:

```
newswap[3, 4]
newswap[3, 4]
```

4.5 Advanced Topic: Scoping

The *scope* of a symbol is the region of a program within which the name of the symbol actually refers to *that* symbol, as opposed to some other symbol having the same name. For example, the scope of a formal parameter in a function is the body of the function; outside the body of the function, the same name refers to a different symbol.

We've already seen three scoping constructs: function declaration, `Module`, and `With`. All three of these constructs scope a symbol in space, that is, in a particular region of the program. This is called *lexical* scoping. In contrast, the often-misunderstood scoping construct `Block` scopes a symbol in *time*; this is called *dynamic* scoping. In Section 4.5.1 we compare `Block` to `Module`. Finally, in Section 4.5.2 we explain how lexical scoping constructs behave when they are nested inside of other lexical scoping constructs.

4.5.1 Lexical versus dynamic scoping

The `Module` command scopes symbols in a way that computer scientists call *lexical scoping* [MacLennan 83]. What this means is that the actual symbol that is bound to a name can be inferred completely from the textual context in which the name is used. Here is an example that hopefully will clarify this explanation:

With this definition, the value of `foo` depends on the value of the global symbol `x`.

```
foo := x;
```

Changing `x` affects the evaluation of `foo`.

```
x = 1; foo
1

x = 2; foo
2
```

But `foo` is unaffected by the local symbol named `x`.

```
Module[{x = 3}, foo]
2
```

Note that `foo` ignores the local symbol `x` and evaluates to the value of the global symbol `x`. In other words, regardless of where `foo` is evaluated, when lexical scoping is in effect, *`foo` always refers to the global symbol `x`*. Lexical scoping arguably makes programs easier to understand, since the names chosen for local symbols do not affect the execution of the program.

In contrast, when *dynamic* scoping is in effect the symbol `x` referred to by `foo` will be the *most recently defined* symbol named `x`. Dynamic scoping is achieved by using the `Block` command:

`foo` refers to the symbol `x` defined by `Block`.

```
Block[{x = 3}, foo]
3
```

Note that the value of the global symbol `x` has not been changed.

```
x
2
```

Put another way, when dynamic scoping is in effect, the symbol `x` referred to by `foo` depends on where `foo` is *evaluated*, rather than where it was *defined*. This may make programs harder to understand.

What `Module[{x}, ...]` does is create a temporary symbol `x` that is distinct from the global symbol (and any other temporary symbols) having the same name.

Every time this command is executed a unique variable name is generated.

```
Module[{x}, x]
x$30
```

As you can see, unlike other procedural programming languages, it is possible to pass these "local" symbols outside of the scope in which they are created, although there's usually no good reason for doing so. The converse is not true: There is no way to refer directly to the global symbol `x` within the scope of this `Module`. The global symbol is *shadowed* by the local symbol.

On the other hand, the `Block` statement shown above is equivalent to the following:

```
savex = x;
x = 3;
foo
x = savex;
3
```

In other words, `Block` creates a temporary *value* for an existing symbol, whereas `Module` creates temporary *symbols* that are distinct from all existing symbols. `Block` is thus not really suitable as a mechanism for declaring local symbols, because it can affect the binding of names even for symbols that are used outside of the `Block`.[8] It is

8. The reason for the ubiquity of `Block` in existing *Mathematica* code is a legacy of the fact that `Module` did not even exist until version 2.0.

most useful for overriding the value of some system-defined global variable for the *duration* of a particular piece of code — hence we say that `Block` scopes in time rather than in space. For example,

`Block` changes the default font used by these plotting functions — and all functions called by them — until `Block` finishes executing.

```
Block[{$DefaultFont={"Times", 17}},
    Plot[...];
    Plot3D[...]
]
```

One argument for using `Block` rather than `Module` or `With` is that `Block` is more efficient. However, the difference in execution time is negligible except for functions that get called an extremely large number of times. If you must use `Block` for this reason, just be sure that you do not inadvertently change the values of any global symbols that are used by other functions involved in the computation. Overriding global symbols used internally by system-defined functions sometimes is a necessary evil. However, in the author's opinion, it is *egregious* programming style to use `Block` to override global symbols used by your *own* functions — that's what parameters are for!

As an aside, note that the semantics of iteration variables in functions such as `Table`, `Sum`, etc. are identical to the semantics of `Block` variables. This is desirable, since otherwise computations such as the following wouldn't work as expected:

```
expr := j^2;
j = 2;
Table[expr, {j, 1, 4}]
{1, 4, 9, 16}
```

Exercise

1. Explain the discrepancy between the results of the following two `Table` commands:

```
Clear[a, i];
a = i; i = 4;

Table[a, {i, 1, 2}]
{1, 2}

Module[{i}, Table[a, {i, 1, 2}]]
{4, 4}
```

4.5.2 Nested scoping constructs

When lexical scoping constructs with possibly conflicting name declarations are nested, the innermost construct that declares a given name is the one that determines the symbol to which the name refers. For example,

Two different `x`'s are being used to compute the result.

```
With[{x = a}, With[{x = b}, x] + 2 x]
2 a + b
```

This corresponds to the way in which local variables behave in a *block-structured* language such as C or Pascal.[9] The general rule is that outer scoping constructs "respect" declarations made in inner scoping constructs.

The previous example works the same way no matter how you mix `Module` and `With`.

```
Module[{x = a}, With[{x = b}, x] + 2 x]
2 a + b
```

Here is a different kind of name conflict.

You may be wondering why the answer is not `3b`.

```
With[{x = a}, With[{a = b}, a + 2 x]]
2 a + b
```

What happened is that the inner `a` was renamed to avoid the conflict.

```
With[{x = a}, Hold[With[{a = b}, a + 2 x]]]
Hold[With[{a$ = b}, a$ + 2 a]]
```

This example illustrates another general principle of *Mathematica*'s lexical scoping constructs: The result should be independent of the names of any temporary variables. In other words, the given example is semantically equivalent to this:

```
With[{x = a}, With[{z = b}, z + 2 x]]
2 a + b
```

Note that scoping applies only to the second argument of a scoping construct, and not to other declarations within the first argument:

The `x` in `y = x` refers to the outer `x`, not the inner one.

```
With[{x = a}, With[{x = b, y = x}, y]]
a
```

Function declaration, too, is a lexical scoping construct. In the following example, even thought the right-hand side of the `Set` command evaluates inside of the `With`, the local constant `y` in the `With` does not affect the formal parameter `y`.

The formal parameter is renamed `y$` to avoid a name conflict.

```
With[{x = 5, y = 6}, g[y_] = x + y]
5 + y$
```

Once again, the explanation is that the result should be the same no matter what name is chosen for the formal parameter to a function.

Scoping issues are covered at great length in §2.6 of *The Mathematica Book*.

---

9. *Note:* Do not confuse the *Mathematica* function `Block` with the term *block* as it is used in regard to block-structured languages (e.g., in C, a sequence of statements surrounded by curly braces; in Pascal, a sequence of statements delimited by `begin...end`). It is an unfortunate accident of the nomenclature that `Module` is the *Mathematica* construct that corresponds to a *block* in block-structured languages.

5

Functional Programming

Chapter 4 illustrated how one can program in a procedural style using *Mathematica*. However, procedural programming constructs in *Mathematica* really are just a facade; deep down inside, *Mathematica* is a *functional* programming language.

In the functional programming paradigm there is no distinction between functions and data. Functions can be manipulated just as any other data, including being passed as arguments to and returned as results from other functions. You can, for example, write a program that writes other programs and executes them. This is why functional programming is so popular in the field of artificial intelligence: It enables the creation of programs that have the ability to change their behavior as they run.

Furthermore, in functional programs there are no assignment statements and no loops.[1] In place of assignments to variables, data are moved strictly through function call and return (by nesting function calls, sometimes quite deeply). In place of loops, there are *higher-order functions* that apply other functions to collections of data, or *recursion* is used. Because functional programming is so different from procedural programming, it does require some effort to master, but the payback is its ability to produce programs that are elegant, concise, and powerful.

In the early parts of this chapter we restrict our attention to performing functional operations on lists. In the last part of the chapter we show how these operations generalize to arbitrary expressions.

The most well-known functional programming language undoubtedly is *Lisp* [Steele 84]. Throughout this chapter we shall remark on the similarities and differences between *Mathematica* and Lisp; a summary of these observations is given in Section 5.7.

1. Although these seem like drastic restrictions, their virtue is that they allow functional programming to be placed on a sound mathematical footing.

5.1 Basic Functional Programming

5.1.1 Map and Apply

`Map` and `Apply` are two of the most fundamental functional programming operations.

`Map` wraps a given function around each element of a list and returns a list of the results.[2] It is similar to the Lisp function `mapcar`.

```
Map[f, {a, b, c}]
{f[a], f[b], f[c]}
```

`Map` has a special input form.

```
f /@ {a, b, c}
{f[a], f[b], f[c]}
```

`Apply`, on the other hand, wraps a function around *all of the elements of a list at once* (not the list itself):

```
Apply[f, {a, b, c}]
f[a, b, c]
```

`Apply`, too, has a special input form.

```
f @@ {a, b, c}
f[a, b, c]
```

For example, suppose we wanted to sum all of the elements of a list. We could do this with some kind of iterator, but it's much more elegant (and efficient) to use `Apply`:

The internal form of the result is `Plus[a, b, c]`.

```
Plus @@ {a, b, c}
a + b + c
```

In contrast, *mapping* `Plus` onto a list has no apparent effect:

```
Map[Plus, {a, b, c}]
{a, b, c}
```

A trace of the evaluation shows why.

Each application of `Plus` is interpreted as unary plus, which is an identity operation.

```
Trace[Map[Plus, {a, b, c}]]
{Plus /@ {a, b, c}, {Plus[a], Plus[b], Plus[c]},
  {Plus[a], a}, {Plus[b], b}, {Plus[c], c}, {a, b, c}}
```

We can use `Apply` to create the following concise function that computes the arithmetic mean of a list. Note that this function works on lists of any length because `Plus` is an *n*-ary operator (Section 2.3.1).

---

2. `Map` was introduced in Section 3.5.3. You may wish to go back and review that material, as well as other coverage of lists in Chapter 3, before proceeding.

```
amean[s_List] := (Plus @@ s) / Length[s]

amean[{a, b, c}]
a + b + c
-----------
    3
```

Now suppose that we want to sum each of the rows of a matrix. Recall that each row of a matrix is stored in a sublist, hence we need to sum each of the sublists of the matrix.

Here's a matrix.

```
mat = Array[a, {3, 3}]
{{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]}}
```

First define a function that sums the elements of a list.

```
sumList[s_] := Plus @@ s

sumList[{a, b, c}]
a + b + c
```

Simply map `sumList` onto the matrix to sum each sublist (i.e., row).

```
Map[sumList, mat]
{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]}
```

As another example, suppose that we had the prime factorization of an integer in the form returned by `FactorInteger`, and that we wanted to "reconstitute" the original integer from it.

Here is the prime factorization of 283500 = $2^2*3^4*5^3*7^1$.

```
factors = FactorInteger[283500]
{{2, 2}, {3, 4}, {5, 3}, {7, 1}}
```

First define a function that exponentiates the first element of a list by the second element of that list.

```
applyPower[s_] := Power @@ s

applyPower[{a, b}]
 b
a
```

Now `Map` this function onto the factorization.

```
Map[applyPower, factors]
{4, 81, 125, 7}
```

Finally, multiply the powers together.

```
Apply[Times, %]
283500
```

Here is a function that performs these steps. Note the use of a `Module` to prevent the auxiliary function from cluttering up the global name space. Auxiliary functions such as `applyPower` (and `sumList` in the previous example) are an annoyance that will be dealt with in the next section.

```
ExpandFactorization[f_] :=
Module[{ap},
    ap[x_] := Power @@ x;
    Times @@ ap /@ f
]
```

The `@@` and `/@` operators have equal precedence and are right-associative, so the expression `Times @@ ap /@ f` is interpreted as `Times @@ (ap /@ f)` or equivalently, `Apply[Times, Map[ap, f]]`. This is an example in which the default grouping of operators corresponds to what is intended; however, as we shall see in the next section, one is not always so lucky. When in doubt, consult the table of special input forms in Appendix A of *The Mathematica Book*, or better yet, use parentheses to settle the matter unambiguously.

Exercises

1. Write a function that computes the largest element in each row of a matrix.
2. Using only `Apply`, `Length`, and built-in numerical functions, define functions to compute the following operations on lists of arbitrary lengths. Remember that all of the arithmetic functions are listable, e.g., `{a, b} + {c, d} == {a + c, b + d}; 1/{a, b} == {1/a, 1/b}`, etc.

   The geometric mean:

```
gmean[{a, b, c}]
```

$$(a\ b\ c)^{1/3}$$

   The harmonic mean:

```
hmean[{a, b, c}]
```

$$\frac{1}{\frac{1}{a} + \frac{1}{b} + \frac{1}{c}}$$

   The dot product:

```
dotprod[{a, b}, {c, d}]
a c + b d
```

   A function that sums the columns of a matrix:

```
colsum[{{a, b}, {c, d}}]
{a + c, b + d}
```

   Euclidean distance between two points in n-dimensional space:

```
euclid[{x1, y1}, {x2, y2}]
Sqrt[(x1 - x2)^2 + (y1 - y2)^2]
```

3. Compare the speed of summing a list by using the iteration functions `For`, `While`, and `Sum` to summing it by using `Apply`.

5.1.2 Pure functions

At this point, we will postpone the introduction of any new functional programming operations to talk about a feature of *Mathematica* that makes all of these operations more convenient to use, namely, *pure functions*.

As savvy *Mathematica* programmers, we are well aware of the fact that there is no real distinction between code and data — everything is an expression. Whether or not an expression is a function is simply a matter of interpretation by the kernel. There are basically only two things that distinguish the body of the function definition `f[x_] := x^(1/3)` from the "naked" expression `x^(1/3)`. First of all, in the former case the body expression is not evaluated until the function is used. Second, the function declaration alerts the kernel that the symbol `x` in the expression is actually a parameter, and not a global symbol called `x`.

It would be nice if we could take an arbitrary expression and say to the kernel, "Please treat this as a function just for a moment," without actually having to define a function in the normal way. It turns out that there is a special head that does exactly that. It is called — what else? — `Function`. The first argument to `Function` is the name of the formal parameter, and the second argument is the expression that we wish to use as a function. Here is an example of using `Function` to create a function that applies `Power` to its argument:

```
Function[x, Power @@ x]
Function[x, Apply[Power, x]]
```

Passing a list to this function applies `Power` to the list.

```
%[{a, b}]
```

$$a^b$$

Note that there is no `Blank` appended to the formal parameter as there is in a normal function definition. Also note that `Function` is a lexical scoping construct (Section 4.5.1); therefore, the name chosen for the formal parameter does not affect the computation.

```
Function[x, Power @@ x][{1 + x, 1/x}]
```

$$(1 + x)^{1/x}$$

An expression with head `Function` is called a *pure function*. Note that `Function` is simply a "container" that provides the twin essentials for interpreting the expression within it as a function: It does not evaluate the expression (it has the `HoldAll` attribute), and it specifies which symbols in the body of the function are parameters.

Also note that `Function` doesn't create any global definitions. Because they have no names, pure functions sometimes are called *anonymous functions*. (In Lisp they are called *lambda expressions*.) If you want to use a pure function more than once, you can assign it to a symbol, just as you would any other expression.

```
Clear[applyPower]
applyPower = Function[x, Power @@ x];
applyPower[{a, b}]
```

$$a^b$$

Now the point of all this is that a `Function` is an expression, so it can be nested inside of any other expression. So, for example, to apply `Power` to every sublist in a list, we could write:

```
Map[Function[x, Power @@ x], {{a, b}, {c, d}}]
```

$$\{a^b, c^d\}$$

Creating a "one time only" function at its point of use not only is convenient for the programmer, but also can make the code easier to understand, because all of the information needed to understand what an expression does is manifested in the expression.

Recall the `ExpandFactorization` function from the previous section. It could be rewritten more elegantly as follows (the `Map` operation has been changed to its long form for clarity):

```
ExpandFactorization[f_] :=
    Times @@ Map[Function[x, Power @@ x], f]
```

There is a special input form for pure functions that makes them very economical to enter. In this syntax, the parameter to the function has no name either; it is referred to as # (pronounced *slot*). The entire expression that is to be turned into a pure function is terminated by the & operator. Using this notation, the definition of the `applyPower` function would be `Power @@ # &`. Here's an example of its use:

```
Map[Power @@ # &, {{a, b}, {c, d}}]
```

$$\{a^b, c^d\}$$

The basic idea is, apply `Power` to *something* — where *something* is supplied later (in this case, by the action of `Map`). Very concise, very elegant, and also very impossible to understand if you haven't seen it before.

Here is the `ExpandFactorization` function yet again, this time using the special input form for the anonymous function.

```
ExpandFactorization[f_] :=
    Times @@ Map[Power @@ # &, f]
```

Note that if we had used the /@ form of the Map function, it would have been necessary to surround the entire anonymous function *including* the & with parentheses, as shown below:

```
Times @@ (Power @@ # &) /@ factors
283500
```

The reason for this is that the & operator has a very low precedence. If the parentheses were omitted, the expression would be interpreted as shown below:

HoldForm prevents the expression from evaluating.

```
HoldForm[Times @@ Power @@ # & /@ factors]
(Times @@ Power @@ #1 & ) /@ factors
```

which is not at all what was intended.[3] One way to think of the action of & is that it collects into the body of the anonymous function the largest complete expression it can find to its left, stopping only when it hits one of the following: a comma, a semicolon, an unmatched left delimiter (parenthesis, square bracket, or curly brace), a // operator, or any assignment operator (e.g., = and :=).

To define a pure function with more than one parameter, simply use a list of formal parameters as the first argument to Function.

```
proportion = Function[{x, y}, x/(x + y)];
proportion[1, 2]
```

$$\frac{1}{3}$$

Pure functions can have more than one anonymous argument; in this case the arguments are referred to as #1, #2, and so on.

This pure function works just like the proportion function defined above.

```
#1/(#1 + #2)&[a, b]
```

$$\frac{a}{a + b}$$

As an aside, note that this section has provided us with our first uncontrived examples of expressions whose heads are themselves normal expressions (rather than symbols).

Exercises

1. Use a pure function to add 1 to every element of a list. Try it using both the long and short input forms for the pure function.
2. Use Partition, Map, and Apply to implement a function that computes a moving average of a list of numbers, i.e.,

3. Another common mistake is to surround the body of the pure function with parentheses but not the &, as in Times @@ (Power @@ #)& /@ factors. Although this *looks* like it ought to work, it doesn't (try it).

```
moveAvg[{a, b, c, d, e}, 2]
```

$$\left\{\frac{a+b}{2}, \frac{b+c}{2}, \frac{c+d}{2}, \frac{d+e}{2}\right\}$$

```
moveAvg[{a, b, c, d, e}, 3]
```

$$\left\{\frac{a+b+c}{3}, \frac{b+c+d}{3}, \frac{c+d+e}{3}\right\}$$

3. Given a string, create a histogram of the frequencies of all characters appearing in the string. *Hints:* Map an anonymous function that counts how many times its argument appears in the string over the list of unique characters in the string. You can break up a string into a list of individual characters using `Characters`. The `Union` function eliminates duplicates from a list. `Count[list, expr]` returns the number of times *`expr`* occurs as an element of *`list`*. Finally, you can plot the histogram using the `BarChart` function from the standard package `Graphics`Graphics``.

5.1.3 Level specifications

The *level* of an element in an expression is the number of subscripts that are necessary to identify its position within the expression. `Level[list, lev]` returns a list of all elements of *`list`* that are at level *`lev`*. For example,

```
alist = {{a, b}, {c, {d, e}}};
```

The sublists `{a, b}` and `{c, {d, e}}` are at level 1.

```
Level[alist, {1}]
{{a, b}, {c, {d, e}}}
```

a, b, c, and the sub-sublist `{d, e}` are at level 2.

```
Level[alist, {2}]
{a, b, c, {d, e}}
```

d and e are at level 3.

```
Level[alist, {3}]
{d, e}
```

By default, `Map` operates only at level `1`, and `Apply` operates only at level `0` (the entire expression). However, both take an optional third argument, called a *level specification*, that controls the levels at which they operate. There are several forms of level specifications, similar to the forms of the index range in functions such as `Take` and `Drop`.

A level specification of the form `{n}` specifies level *n* only. Compare the following results carefully:

The default behavior of `Map` is to operate at level 1 only.

```
Map[f, alist]
{f[{a, b}], f[{c, {d, e}}]}
```

Here we `Map` at level 2.

```
Map[f, alist, {2}]
{{f[a], f[b]}, {f[c], f[{d, e}]}}
```

And here we Apply at level 1.

```
Apply[f, alist, {1}]
{f[a, b], f[c, {d, e}]}
```

Note in particular the rather subtle difference between mapping a function at level 1 (the default) and applying the function at level 1. In fact, `Apply[f, list, {1}]` is equivalent to `Map[f @@ # &, list]`, which makes applying a function at level 1 one of the more common uses of level specifications. For example, here is the `ExpandFactorization` function one last time:

Compare this to the definition of ExpandFactorization on page 102.

```
ExpandFactorization[f_] :=
    Times @@ Apply[Power, f, {1}]
```

(See the exercises for more examples of this technique.) Another common use of level specifications is mapping at level 2, which can be used to map a function onto every element of a matrix. We'll see several examples of this in Section 5.2.2.

There are two other forms of level specifications. A level specification of the form *n* (without the braces) applies to levels 1 through *n* inclusive. (*Exception:* A level specification of 0 doesn't do anything.)

```
Map[f, alist, 2]
{f[{f[a], f[b]}], f[{f[c], f[{d, e}]}]}
```

You can use Infinity to mean all levels greater than 0.

```
Map[f, alist, Infinity]
{f[{f[a], f[b]}], f[{f[c], f[{f[d], f[e]}]}]}
```

A level specification of the form `{n, m}` applies to levels *n* through *m* inclusive:

```
Map[f, alist, {2, 3}]
{{f[a], f[b]}, {f[c], f[{f[d], f[e]}]}}
```

The function `MapAll` is equivalent to `Map` with a level specification of `{0, Infinity}`: It wraps a function around *every* part of the list, including the entire list itself.

```
MapAll[f, alist]
f[{f[{f[a], f[b]}], f[{f[c], f[{f[d], f[e]}]}]}]
```

Note that when a function is mapped or applied at several (or all) levels, the process is carried out in a depth-first fashion. That is, the operation is done at lower levels before it is done at higher ones.

The elements are printed in the order that they are "visited" by Map.

```
Map[(Print[#]; f[#])&, alist, {1, 2}];
a
b
{f[a], f[b]}
c
{d, e}
{f[c], f[{d, e}]}
```

Level numbers also can be negative. As you might expect, negative levels are counted "up from the bottom," but the concept of "bottom" is probably not what you expect.

All atoms are at level -1.

```
Level[alist, {-1}]
{a, b, c, d, e}
```

Counting up from the bottom corresponds to including larger and larger sublists.

```
Level[alist, {-2}]
{{a, b}, {d, e}}

Level[alist, {-3}]
{{c, {d, e}}}
```

Note that in an unbalanced list structure, the elements at level $-n$ may come from many different (positive) levels. If you think of a nested list as a tree in which the entire list is the root and the atoms are the leaves, then positive levels are counted downward from the root whereas negative levels are counted upward from the leaves. This is illustrated in Figure 5-1.

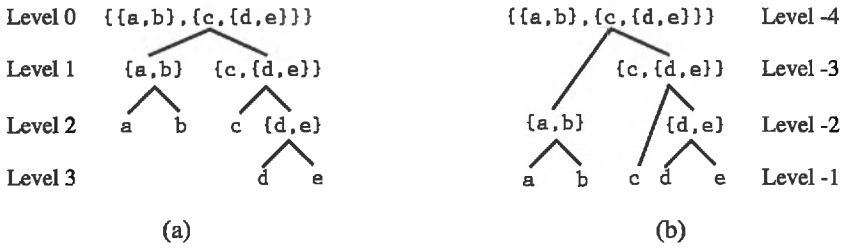


Figure 5-1 (a) Levels in a nested list. (b) Negative levels in the same list.

Table 5-1 lists all of the functions that accept level specifications. Table 5-2 lists additional functions that allow level specification in nonstandard ways (look at some of the usage messages for examples). All of the functions in these tables are covered in this chapter, Chapter 3, "Lists and Strings," or Chapter 6, "Rule-Based Programming."

Table 5-1 Functions that accept standard level specifications.

Apply	Cases	Count	DeleteCases
FreeQ	Level	Map	MapIndexed
MemberQ	Position	Scan	

Table 5-2 Functions that use nonstandard level specifications.

Dimension	Flatten	MapThread	Operate
Outer	Partition	RotateLeft	RotateRight
Transpose	TreeForm		

Exercises

1. Using only `Apply` and `Plus`, write a function that sums the rows of a matrix.

```
rowsum[{{a, b}, {c, d}}]
{a + b, c + d}
```

2. Rewrite Exercise 5.1.2.2 without using `Map`.

5.1.4 MapAt

Contrast the use of `Map` with level specifications to the function `MapAt`, which allows you to target the application of a function to a *particular element* within an expression rather than to an entire level. For example,

`b` is at position `{1, 2}`.

```
Position[alist, b]
{{1, 2}}
```

Therefore, this wraps `f` only around `b`.

```
MapAt[f, alist, {1, 2}]
{{a, f[b]}, {c, {d, e}}}
```

Multiple specifications can be enclosed within another list.

```
MapAt[f, alist, {{1, 2}, {2, 2, 1}}]
{{a, f[b]}, {c, {f[d], e}}}
```

We'll see some sophisticated uses of `MapAt` in Section 5.5.3.

There is no corresponding function `ApplyAt`; however, it's not hard to build one using `MapAt`.

Exercise

1. Write an `ApplyAt` function. It should behave as follows:

```
ApplyAt[f, alist, {{1}, {2, 2}}]
{f[a, b], {c, f[d, e]}}
```

5.2 Variations on a Theme

`Map` has quite a few less-familiar relatives that are useful in particular circumstances.

5.2.1 MapThread and Thread

`MapThread` is a generalization of `Map` to functions that take more than one argument. The first argument to `MapThread` is the function to be mapped, and the second argument is a matrix.

```
MapThread[f, {{a1, a2}, {b1, b2}, {c1, c2}}]
{f[a1, b1, c1], f[a2, b2, c2]}
```

In other words, MapThread applies the function to each *column* of the matrix. (This could also be achieved using Transpose followed by Apply, but MapThread is much more efficient.)

The equivalent effect can be achieved with Thread, but first the function to be threaded has to be applied to the arguments to be threaded over.

```
Thread[f[{a1, a2}, {b1, b2}, {c1, c2}]]
{f[a1, b1, c1], f[a2, b2, c2]}
```

MapThread is somewhat limited by the restriction that the second argument must be a matrix. Here's what happens if it is not:

```
MapThread[f, {{a1, a2}, {b1, b2}, c}]
MapThread::mptd:
   Object c at position {2, 3} in
    MapThread[f, {{a1, a2}, {b1, b2}, c}] has only 0
     of required 1 dimensions.
MapThread[f, {{a1, a2}, {b1, b2}, c}]
```

Thread is more flexible — it allows any of the arguments to be scalars.

```
Thread[f[{a1, a2}, {b1, b2}, c]]
{f[a1, b1, c], f[a2, b2, c]}
```

In fact, Thread is the mechanism that underlies listability (Section 3.3):

```
?Listable
Listable is an attribute that can be assigned to a
   symbol f to indicate that the function f should
   automatically be threaded over lists that appear as
   its arguments.
```

Suppose that you have a list of values that you want to turn into rules so that they may be substituted into an equation.

Here's one way a list of values could arise.

```
LinearSolve[Table[Random[], {3}, {3}],
            Table[Random[], {3}]]
{0.857971, -0.191116, -0.01025}
```

Rule is not listable.

```
{x1, x2, x3} -> %
{x1, x2, x3} -> {0.857971, -0.191116, -0.01025}
```

But it can be "persuaded" by using Thread.

```
Thread[%]
{x1 -> 0.857971, x2 -> -0.191116, x3 -> -0.01025}
```

There is one important caveat about using `Thread`, however: It evaluates its argument before doing the threading. Here's an example showing the consequences of this. Suppose you want to perform a pairwise equality test on two lists.

`Equal` is not `Listable`; it compares the two lists in their entirety.

```
{1, 2, 3} == {1, 2, 4}
False
```

`MapThread` works fine.

```
MapThread[Equal, {{1, 2, 3}, {1, 2, 4}}]
{True, True, False}
```

`Thread` does not.

```
Thread[Equal[{1, 2, 3}, {1, 2, 4}]]
Thread::normal:
   Normal expression expected at position 1 in
    Thread[False].
Thread[False]
```

What happened here is that `Equal[{1, 2, 3}, {1, 2, 4}]` evaluated to `False` before `Thread` was even called.

Here's a trick for getting around this type of problem (which occurs in many circumstances) that every *Mathematica* programmer should know about. `Unevaluated` can be wrapped around any argument to a function to pass the unevaluated form of the argument to the function:

```
Thread[Unevaluated[Equal[{1, 2, 3}, {1, 2, 4}]]]
{True, True, False}
```

`Unevaluated` is analogous to the Lisp special form `quote`. We will discuss `Unevaluated` in detail in Section 7.2.4.

Exercises

1. Use `Thread` and `Apply` to write a function called `myMapThread` that works even if some of the elements in the second argument to `MapThread` are scalars, i.e.,

```
myMapThread[Power, {{a, b, c}, 3}]
{a^3, b^3, c^3}
```

2. Does your `myMapThread` function work in the following case?

```
myMapThread[Equal, {{1, 2, 3}, {1, 2, 4}}]
{True, True, False}
```

3. You can generate a list of all listable functions as follows. First, generate a list of the names of all system-defined symbols using `Names["System`*"]` (this list will be very large). Next, note that `Attributes["`*symname*`"]`, returns the list of attributes for the symbol *symname*. Thus, `MemberQ[Attributes["`*symname*`"], Listable]` returns `True` if the symbol *symname* is listable. Now you can extract

the names of the listable symbols from the list of all system-defined symbol names by using `Select` with an appropriate pure function.

5.2.2 MapIndexed

Sometimes the function you want to map onto a list needs to know the position of the element it is operating upon. For example, you may want to compare every element in the list to its neighbor(s), compute a moving average of data, or check a matrix to see if it has any special structure. You might think that you have to resort to an explicit loop to do this (so that the function has access to the loop index), but in fact there are alternatives. The `MapIndexed` function is one such alternative.

`MapIndexed` applies a given function to each element of a list, but passes an additional argument to the function indicating the position of the list element. Here's a simple example:

The second argument passed to the function contains the position of the current element.

```
MapIndexed[f, {a, b, c}]
{f[a, {1}], f[b, {2}], f[c, {3}]}
```

Of course, the actual function being mapped had better expect this additional argument.

The position argument is wrapped in list braces so that `MapIndexed` can work at deeper levels of the list in a consistent manner:

The level specification maps `f` at level 2 of this list. Thus every position is a list of two integers.

```
MapIndexed[f, {{a, b, c}, {d, e}}, {2}]
{{f[a, {1, 1}], f[b, {1, 2}], f[c, {1, 3}]},
  {f[d, {2, 1}], f[e, {2, 2}]}}
```

One practical use of `MapIndexed` is to check a matrix for certain properties. For example, suppose we wanted to check a matrix to see if it is diagonal. For any element `x` at position `{i, j}` in the matrix, either `x` must be zero or `i` must equal `j`. The latter condition can be checked easily by using `Equal @@ {i, j}`. Therefore all we need to do is to use `MapIndexed` to map the pure function `Function[{x, index}, x == 0 || Equal @@ index]` onto the matrix at level 2:

This matrix is diagonal.

```
m1 = {{1, 0}, {0, 1}};

MapIndexed[
    Function[{x, index}, x == 0 || Equal @@ index],
    m1, {2}]
{{True, True}, {True, True}}
```

This matrix has a nonzero entry off the diagonal at position `{1, 2}`.

```
m2 = {{1, 1}, {0, 1}};
```

Here we use the special input syntax for the pure function.

```
MapIndexed[#1 == 0 || Equal @@ #2 &, m2, {2}]
{{True, False}, {True, True}}
```

It's hard to imagine a more elegant way of doing this. Note that a procedural solution would require a nested loop!

We're not quite finished: We've produced a matrix of boolean values, but what we really want is a single value, `True` or `False`. Applying the And function to the booleans seems to be the obvious course of action:

First `Flatten` the matrix into a list.

```
Flatten[%]
{True, False, True, True}
```

Now apply `And`.

```
And @@ %
False
```

Here is our new predicate, `DiagonalQ`.

```
DiagonalQ[m_] :=
    And @@ Flatten[
        MapIndexed[#1 == 0 || Equal @@ #2 &, m, {2}]
    ]
```

The function being mapped can also use the information in the second argument to do things like access neighboring elements. Here's an example in which we average each element with its neighbor(s) to the right.

This function computes the average of `s[[i]], ..., s[[i + width - 1]]`.

```
avgrt[s_, x_, index_, width_] :=
With[{i = index[[1]]},
    If[ i + width - 1 <= Length[s],
        (Plus @@ Take[s, {i, i + width - 1}]) / width,
        x]
]
```

Note that the list has to be passed to `avgrt` explicitly; `MapIndexed` doesn't pass it for you.

```
z = {a, b, c, d, e, f};
MapIndexed[avgrt[z, #1, #2, 2]&, z]
```

$$\left\{\frac{a+b}{2}, \frac{b+c}{2}, \frac{c+d}{2}, \frac{d+e}{2}, \frac{e+f}{2}, f\right\}$$

It must be pointed out that for moving averages, there may be superior alternatives to `MapIndexed`. (See Exercise 5.1.2.2 for one such alternative.)

Exercises

1. Use `MapIndexed` to write a function that takes a matrix and a scalar value as arguments, and returns a new matrix that is identical to the original one except that all of its diagonal elements are set to the given scalar, e.g.,

```
setdiag[{{a, b, c}, {d, e, f}, {g, h, i}}, 0]
{{0, b, c}, {d, 0, f}, {g, h, 0}}
```

2. Use `MapIndexed` to write predicates called `TriDiagonalQ`, `LowerTriangularQ`, and `UpperTriangularQ` that test a matrix for the corresponding properties.
3. The notion of "diagonality" is easy to extend to higher dimensions. Rewrite `DiagonalQ` so that it works on matrices of any dimension. *Hint:* Map the anonymous function at level `-1`.
4. A *Toeplitz* matrix is an $n \times n$ matrix (a_{ij}) such that $a_{ij} == a_{i-1,j-1}$ for i and j in the range $2 \ldots n$. Write a predicate to check for this property.

5.2.3 Through

`Through` is complementary to `Thread`: Rather than applying a single function to arguments that are lists, `Through` applies a *list of functions* to scalar arguments. Compare the following to see the difference:

```
Thread[f[{a1, b1, c1}, {a2, b2, c2}]]
{f[a1, a2], f[b1, b2], f[c1, c2]}

Through[{f, g, h}[a, b, c]]
{f[a, b, c], g[a, b, c], h[a, b, c]}
```

`Through` is rather uncommon; we'll see just a few examples of its use in this book.

5.2.4 Scan

`Scan` is not a true functional primitive. It is like `Map` in that it applies a function to a list of arguments, but it does not return a list of the results. By default, `Scan` returns `Null`, which means that it can act only through side effects:

```
Scan[Print, {a, b}]
a
b
```

An example of a useful side effect would be to increment a counter, perhaps to create a histogram of the values in a list. For example,

Here are some data.

```
data = Table[Random[], {100}];
```

Initialize a histogram with 5 bins.

```
hist = Table[0, {5}];
```

Since the data are real numbers between `0` and `1`, `Ceiling[5#]` is a number between `1` and `5`.

```
Scan[hist[[Ceiling[5#]]]++ &, data]
```

Here's the result.

```
hist
{17, 23, 21, 13, 26}
```

Scan is made more useful by the fact that if the function being applied returns a value explicitly (using Return), Scan will terminate and return that value.[4] This behavior makes Scan more efficient than Map for such tasks as determining if all elements of a list satisfy a predicate. Compare the following:

This function leaves an "audit trail" as it tests for oddness.

```
myOddQ[x_] := (Print[{x, OddQ[x]}]; OddQ[x])
```

The function is applied to every element of the list. A procedural version of this would probably halt after encountering the 2.

```
And @@ myOddQ /@ {1, 2, 3, 4, 5}
{1, True}
{2, False}
{3, True}
{4, False}
{5, True}
False
```

The function being scanned executes a Return, causing Scan to terminate early.

```
Scan[If[myOddQ[#], Null, Return[False]]&,
     {1, 2, 3, 4, 5}]
{1, True}
{2, False}
False
```

Using Scan correctly is slightly tricky, however, since if no Return statements are encountered, Scan returns Null:

```
Scan[If[myOddQ[#], Null, Return[False]]&, {1, 3, 5}]
{1, True}
{3, True}
{5, True}
```

We would like to return True under such circumstances. This will be fixed in the exercises.

Exercises

1. Use Scan to write a function called alltrue that takes a predicate and a list as arguments and returns True if all elements of the list satisfy the predicate, False otherwise. Be sure to fix the problem noted in the last example.
2. Use Scan to write a function called anytrue that takes the same arguments as alltrue but tests to see if *any* element in the list satisfies the predicate.

4. In contrast, virtually every other system-defined function is unaffected by Return. For an example of the consequences, see Section 5.3.4.

5.3 Iterating Functions

Mathematica provides three functions for iterating other functions, either for a given number of times or until some convergence criterion is satisfied. These functions frequently can be used in place of procedural looping constructs.

5.3.1 Nest

The `Nest` function composes a function with itself a given number of times, starting from a specified value.

```
Nest[f, a, 4]
f[f[f[f[a]]]]
```

Here is an example of using `Nest` to compute a continued fraction.

```
Nest[1/(1 + #) &, x, 3]
```

$$\frac{1}{1+\frac{1}{1+\frac{1}{1+x}}}$$

It might be easier to understand what's going on if we could see the intermediate expressions that are used to build up the final expression. That is what `NestList` is for.

```
NestList[1/(1 + #) &, x, 3]
```

$$\left\{x, \frac{1}{1+x}, \frac{1}{1+\frac{1}{1+x}}, \frac{1}{1+\frac{1}{1+\frac{1}{1+x}}}\right\}$$

Exercise

1. Use `NestList` to calculate a random walk[5] on the integers. That is, starting from some initial value, repeatedly move up or down by 1 with a 50-50 (or some other) chance. You can plot the result using `ListPlot`.

5.3.2 FixedPoint

The function `FixedPoint` is like `Nest` except that it continues to compose the given function until the returned values converge or until an optional iteration limit is reached. This is useful for performing iterative numerical procedures. Here is an exam-

5. Random walks often are used to model gambling games. The initial value is the amount of money with which the gambler begins. The problem of determining how long, on average, until this value reaches 0 is called the *Gambler's Ruin* problem.

ple in which we use `FixedPoint` to implement a well-known iterative computation for approximating the square root of 2:

```
FixedPoint[(# + 2/#)/2 &, 1.]
1.41421
```

Note the decimal point in the second argument; this is *very important*, for reasons explained below.

As with `NestList`, there's an analogous function called `FixedPointList` that shows all of the intermediate results.

This computation converges quite rapidly.

```
FixedPointList[(# + 2/#)/2&, 1.]
{1., 1.5, 1.41667, 1.41422, 1.41421, 1.41421, 1.41421}
```

As noted above, using an approximate number in at least one of the arguments is *crucial*. It prevents symbolic expansion of the computation, which would be a mess — and wouldn't converge either. The third argument to `FixedPointList` in the following example specifies an iteration limit. Without it, this computation would continue until *Mathematica* had run out of memory.

```
FixedPointList[(# + 2/#)/2&, 1, 5]
{1, 3/2, 17/12, 577/408, 665857/470832, 886731088897/627013566048}
```

The reason that this computation would fail to terminate without the iteration limit is that the default test for convergence is syntactic, not semantic. Even though the *numerical* values of the rationals are converging, the rationals themselves remain syntactically distinct.

The `SameTest` option can be used to specify a different convergence test, which should be a function of two arguments that returns `True` or `False`. This is particularly useful if the iterates are not approximate numbers (e.g., they may be vectors or symbolic quantities). For example, suppose we were interested in a rational approximation to the square root of 2 to within a given tolerance. We could proceed this way:

This computation converges before the iteration limit is reached.

```
FixedPointList[(# + 2/#)/2&, 1, 20,
                SameTest->(Abs[N[#1 - #2]] < 10^-12&)]
{1, 3/2, 17/12, 577/408, 665857/470832, 886731088897/627013566048,
  1572584048032918633353217/1111984844349868137938112}
```

It is always prudent to specify an iteration limit to prevent infinite looping in case the function being iterated does not converge. Even when you "know" the computation converges, it is still all too easy to make a typographic error (such as leaving out a dec-

imal point) that prevents convergence. If this should happen, you can usually abort the computation without killing the kernel.[6]

Exercise

1. Newton discovered that for any "well-behaved" function f, a root of the equation $f(x) = 0$ can be approximated by the following iterative formula:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

Implement Newton's method using `FixedPoint`. (It would be prudent to specify an iteration limit.) Use it to find a root of the `BesselJ[0, x]` function. Try different starting points and see what happens.

5.3.3 Fold

The function `Fold` is somewhat like `Nest`, except that its first argument is a function of *two* arguments. It is similar to the `reduce` function of Lisp. Here is the analogous function `FoldList`, which makes the operation of `Fold` easier to understand:

The list of second arguments to `f` gets "folded in" as the computation proceeds.

```
FoldList[f, a, {b, c, d}]
{a, f[a, b], f[f[a, b], c], f[f[f[a, b], c], d]}
```

Although this seems like a curiosity, it's actually very useful. For example, if `f` is the `Plus` function, `FoldList` will compute all cumulative partial sums of the elements of a list:

```
cumulsum[l_] := FoldList[Plus, First[l], Rest[l]]
cumulsum[{a, b, c, d}]
{a, a + b, a + b + c, a + b + c + d}
```

This technique will work for any left-associative operator.

Another practical use of `Fold` is to implement *Horner's rule* for evaluating a polynomial. The basic motivation behind Horner's rule is that it significantly reduces the number of multiplications used to evaluate the polynomial. For example, Horner's rule would compute the polynomial $a_0 + a_1x + a_2x^2 + a_3x^3$ as $a_0 + x(a_1 + x(a_2 + xa_3))$. This can be accomplished very elegantly using `Fold`:

```
Fold[(#1*x + #2)&, a3, {a2, a1, a0}]
a0 + x (a1 + x (a2 + a3 x))
```

In Section 5.3.5 we will use `Fold` to implement a finite state machine.

---

6. Press **Command-.** (period) on the MacOS, **Alt-.** on Windows, or **Control-C** on UNIX. Consult the *User's Guide* for your specific system if you are unsure.

Exercises

1. Horner's rule is exactly the right tool for converting numbers from arbitrary bases into base 10. For example,

$$\begin{aligned} 10110_2 &= 1\cdot 2^4 + 0\cdot 2^3 + 1\cdot 2^2 + 1\cdot 2^1 + 0\cdot 2^0 \\ &= ((((1\cdot 2+0)\cdot 2+1)\cdot 2+1)\cdot 2+0) \\ &= 22_{10} \end{aligned}$$

   Write a function that takes a list of digits and a base (≤10) as arguments and returns the equivalent base 10 number.[7]

2. Write a function `pdf2cdf` that takes a discrete probability density of the form `{{x1, prob[x1]}, {x2, prob[x2]}, ...}` and returns the cumulative distribution function `{{x1, prob[x1]}, {x2, prob[x1] + prob[x2]}, ...}`. *Hint:* Use `Transpose`.

5.3.4 Throw and Catch

`Throw` and `Catch` provide a mechanism for altering the flow of execution of any *Mathematica* computation. Recall from Section 4.3.4 that *Mathematica* contains functions such as `Break`, `Continue`, and `Return` for altering or terminating the execution of loops and procedures. Unfortunately, these functions cannot be used to exit from any of the functional iteration constructs discussed in this section. For example, when the function `f`, defined below, is passed to `NestList`, the `Return` expression within `f` has no apparent effect on the computation:

```
f[x_] := (If[x > 5, Return[x + 1]]; 2x)
NestList[f, 1, 6]
{1, 2, 4, 8, 9, 10, 11}
```

The reason for this behavior is that the `Return` causes `f`, not `NestList`, to return. It would be useful if there were some way to return from more than one function at a time. This is exactly what `Throw` and `Catch` are for.

The expression `Catch[`*`expr`*`]` normally returns the value of *`expr`*. However, if a `Throw[`*`val`*`]` is encountered during the evaluation of *`expr`*, the evaluation stops and `Catch[`*`expr`*`]` returns *`val`* instead. Here is a concrete example:

Redefine `f` to use `Throw` rather than `Return`.

```
Clear[f]
f[x_] := (If[x > 5, Throw[x + 1]]; 2x)
```

In this case, the `Throw` is not executed.

```
Catch[NestList[f, 1, 3]]
{1, 2, 4, 8}
```

7. This operation is so frequently requested that version 3.0 introduced a new function, `FromDigits`, that performs it.

Here, `Throw[8 + 1]` is executed.

```
Catch[NestList[f, 1, 6]]
9
```

`Throw` and `Catch` are completely general; there is no limit to how "far" (i.e., in terms of the number of nested function returns) a value can be thrown.

In version 3.0, `Throw` takes an optional second argument called a *tag*, and `Throw[`*`val`*`, `*`tag`*`]` is caught only by `Catch[`*`expr`*`, `*`form`*`]`, where *`form`* matches *`tag`*.[8] This allows you to use `Throw` and `Catch` to implement *exception handling* with *named exceptions*. See [Wolfram 96] §2.5.9 for more information.

Exercise

1. The `FindRoot` function takes optional arguments that specify bounds on the range of acceptable values for a solution; if the iterates ever leave this range, `FindRoot` terminates early. Use `Throw` and `Catch` to add this functionality to the Newton's method function that you implemented in Exercise 5.3.2.1.

5.3.5 Application: Huffman coding

In this section we apply functional programming techniques to the problem of constructing and using *Huffman codes* (described below). This application provides many excellent examples of the use of high-level functional programming operations, such as `FixedPoint` and `Fold`, and the treatment of functions as data. It also demonstrates how *Mathematica* lists can be used to represent graph-theoretic trees. This section is quite long but should reward the reader by crystallizing many important concepts.

Character codes

A *character code* is a mapping from characters to binary integers. Character codes come in two basic "flavors," fixed-length and variable-length. The advantage of variable-length codes is that they typically can encode a particular message in fewer bits than a fixed-length code. A Huffman code is a particular type of variable-length character code that has several desirable properties. Before getting into the details of Huffman codes, however, we will give an example to motivate their development. Here's an example message that we wish to encode:

```
msg = "she sells sea shells by the sea shore";
```

In preparation for constructing a code we have to find the *alphabet* of the message, i.e., the set of all unique characters that appear in the message.

8. The *`form`* argument can be a constant expression or it can be a *pattern* (Section 6.1).

First, break up the message into characters.

```
chars = Characters[msg]
{s, h, e,  , s, e, l, l, s,  , s, e, a,  , s, h, e,
  l, l, s,  , b, y,  , t, h, e,  , s, e, a,  , s, h,
  o, r, e}
```

`Union` eliminates duplicates.

```
alphabet = Union[chars]
{ , a, b, e, h, l, o, r, s, t, y}
```

We can create a fixed-length character code as follows:

Here are all 4-bit binary integers between 1 and 11.

```
fcodes = Table[IntegerDigits[i, 2, 4], {i, 11}]
{{0, 0, 0, 1}, {0, 0, 1, 0}, {0, 0, 1, 1},
  {0, 1, 0, 0}, {0, 1, 0, 1}, {0, 1, 1, 0},
  {0, 1, 1, 1}, {1, 0, 0, 0}, {1, 0, 0, 1},
  {1, 0, 1, 0}, {1, 0, 1, 1}}
```

Convert the codes to a set of rules.

```
InputForm[frules = Thread[alphabet->fcodes]]
{" " -> {0, 0, 0, 0}, "a" -> {0, 0, 0, 1},
  "b" -> {0, 0, 1, 0}, "e" -> {0, 0, 1, 1},
  "h" -> {0, 1, 0, 0}, "l" -> {0, 1, 0, 1},
  "o" -> {0, 1, 1, 0}, "r" -> {0, 1, 1, 1},
  "s" -> {1, 0, 0, 0}, "t" -> {1, 0, 0, 1},
  "y" -> {1, 0, 1, 0}}
```

Embed the rules in a function called `fixedCode`.

```
fixedCode[c_String] := c /. frules
```

Here is the entire fixed code.

```
TableForm[{InputForm[#], {fixedCode[#]}}& /@ alphabet]
" "   0 0 0 0
"a"   0 0 0 1
"b"   0 0 1 0
"e"   0 0 1 1
"h"   0 1 0 0
"l"   0 1 0 1
"o"   0 1 1 0
"r"   0 1 1 1
"s"   1 0 0 0
"t"   1 0 0 1
"y"   1 0 1 0
```

Here's a general-purpose encoding function that will work with any character code function such as `fixedCode`:

```
encode[code_, msg_] :=
    Flatten[code /@ Characters[msg]]
```

Encode the message using `fixedCode`.

```
encode[fixedCode, msg] // Short
{1, 0, 0, 0, 0, 1, 0, 0, 0, <<135>>, 0, 0, 1, 1}
```

There are `Length[msg]*4` bits in the encoded message.

```
Length[%]
148
```

At this point a real encoding function would pack the bits into bytes to save space. This is quite a chore to do in *Mathematica* (see the exercises). It's much easier to do this sort of "bit bashing" in a lower-level language like C; we will do exactly that in Chapter 11.

Contrast the fixed-length code with the following variable-length code:

The `variableCode` function will be developed later in this section.

```
TableForm[
    {InputForm[#], {variableCode[#]}}& /@ alphabet]
" "   1 1 0
"a"   0 0 0 1
"b"   1 0 1 0 0
"e"   1 1 1
"h"   0 0 1
"l"   1 0 0
"o"   1 0 1 0 1
"r"   1 0 1 1 0
"s"   0 1
"t"   1 0 1 1 1
"y"   0 0 0 0
```

This variable-length code is an example of a Huffman code. Note that the most common character in the message, `"s"`, has a 2-bit code, whereas the least common characters have 5-bit codes. It can be shown that a Huffman code is optimal *for a given body of text* in the sense that no other character code, fixed- or variable-length, can encode the given text in fewer bits. For this particular message the Huffman code requires about 23 percent less space to encode the message than does the fixed-length code.

```
Short[encodedText = encode[variableCode, msg]]
{0, 1, 0, 0, 1, 1, 1, 1, 1, <<101>>, 0, 1, 1, 1}

Length[encodedText]
114
```

Constructing a Huffman code

The algorithm used to construct a Huffman code can be found in many books on computer algorithms and/or coding theory; one such source is [Cormen et al. 90]. We illustrate this algorithm next.

First, each character in the alphabet, along with the number of times it appears in the message, is placed in a "pool."

```
pool = {Count[chars, #], #}& /@ alphabet
{{7,  }, {2, a}, {1, b}, {7, e}, {4, h}, {4, l},
  {1, o}, {1, r}, {8, s}, {1, t}, {1, y}}
```

Now we need to do the following repeatedly: Find the two elements in the pool having the smallest counts, combine them, and return the combined element to the pool.

Sort the pool.

```
pool = Sort[pool];
```

Here are the two elements with the smallest counts.

```
Take[pool, 2]
{{1, b}, {1, o}}
```

Separate the counts from the characters.

```
Transpose[%]
{{1, 1}, {b, o}}
```

Now apply `Plus` to the sublist containing the counts.

```
MapAt[Plus @@ # &, %, {1}]
{2, {b, o}}
```

Remove the two original elements from the pool and return the combined element. Note that `Union` sorts its result.

```
Union[Drop[pool, 2], {%}]
{{1, r}, {1, t}, {1, y}, {2, a}, {2, {b, o}}, {4, h},
  {4, l}, {7,  }, {7, e}, {8, s}}
```

Putting it all together, we have the following function for replacing the two elements in the pool having the smallest counts with a combined element whose count is the sum of the original two counts:

```
combine[x_] :=
    If[ Length[x] < 2, x,
        Union[Drop[x, 2],
              {MapAt[Plus @@ # &,
                     Transpose[Take[x, 2]],
                     {1}]}
        ]
    ]
```

`b` and `o` are combined.

```
pool = combine[pool]
{{1, r}, {1, t}, {1, y}, {2, a}, {2, {b, o}}, {4, h},
  {4, l}, {7,  }, {7, e}, {8, s}}
```

`r` and `t` are combined.

```
pool = combine[pool]
{{1, y}, {2, a}, {2, {b, o}}, {2, {r, t}}, {4, h},
  {4, l}, {7,  }, {7, e}, {8, s}}
```

`y` and `a` are combined.

```
pool = combine[pool]
{{2, {b, o}}, {2, {r, t}}, {3, {y, a}}, {4, h},
  {4, l}, {7,  }, {7, e}, {8, s}}
```

`{2, {b, o}}` and `{2, {r, t}}` are combined to form `{4, {{b, o}, {r, t}}}`.

```
pool = combine[pool]
{{3, {y, a}}, {4, h}, {4, l}, {4, {{b, o}, {r, t}}},
  {7,  }, {7, e}, {8, s}}
```

We think of `{{b, o}, {r, t}}` as a binary tree whose left subtree is a tree with leaves `b` and `o` and whose right subtree is a tree with leaves `r` and `t`. The integer in the first position of the structure is the total number of occurrences in the message of any of the leaves in the tree — the *weight* of the tree. The algorithm is thus combining trees with smaller weights into trees with larger weights. We can continue this process to its conclusion using `FixedPoint`. (Note that the process terminates because `combine` returns its argument unchanged when the argument is a single-element list.) The result is a single tree whose total weight is the number of characters in the entire message:

```
FixedPoint[combine, pool]
{{37, {{{{y, a}, h}, s},
    {{1, {{b, o}, {r, t}}}, { , e}}}}}

tree = %[[1, 2]]
{{{{y, a}, h}, s}, {{1, {{b, o}, {r, t}}}, { , e}}}
```

A graphical representation of this tree is shown in Figure 5-2. Note that it is a full binary tree (i.e., every internal node has exactly two children), and that the more common characters are higher in the tree than the less common ones.

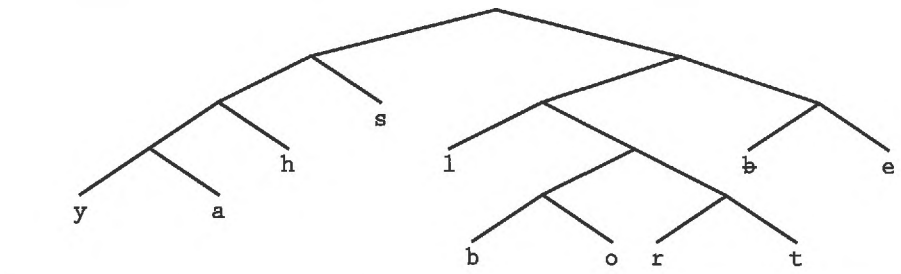


Figure 5-2 A Huffman code tree. (␣ is the blank character.)

The tree so constructed is used to generate a code as follows. Each branch in the tree is labeled with a `0` if it is the left branch out of its parent node and with a `1` if it is the right branch. The concatenation of the branch labels along the path from the root of the tree to any leaf is used as the code for that leaf. For example, to get to leaf `h`, we go left, left again, and then right. Hence, the code for `h` is `{0, 0, 1}`. It can be shown (see [Cormen et al. 90]) that the code produced by this construction is optimal.

These codes can be extracted from the tree automatically. First, note that the position of any leaf in the tree is a list consisting of only `1`'s and `2`'s, since each subexpression in the tree has two parts.

```
Position[tree, "h"]
{{1, 1, 2}}
```

If we subtract `1` from each index, we get the code for that leaf.

```
%[[1]] - 1
{0, 0, 1}
```

This observation leads to the following definition of the `variableCode` function used earlier.

```
variableCode[c_String] := Position[tree, c][[1]] - 1
```

Decoding a coded message

Finding the character corresponding to a binary character code is a simple matter of using the code to index into the code tree.[9]

First add 1 to each bit.

```
{1, 0, 1, 0, 0} + 1
{2, 1, 2, 1, 1}
```

```
Extract[tree, %]
b
```

Here's a function that combines these steps.

```
code2char[t_List, c_List] := Extract[t, c + 1]
code2char[tree, {1, 0, 1, 0, 0}]
b
```

There's only one problem: Since the code is variable-length, we don't know where in the list of binary digits the individual character codes begin and end. Fortunately, Huffman codes have another useful property: They are *prefix-free*. Prefix-free means that no valid code is a prefix of any other valid code.[10] Therefore, we need to "walk" through the code tree as the encoded message is scanned, branching to the left or right according to whether the next bit is a 0 or a 1. When we reach a leaf of the tree, we output the corresponding character and start again at the root of the tree. We can think of this as the operation of a finite state machine in which the code tree is a sort of transition table and the current position within the tree is the state of the machine.

Think about how a finite state machine works: There is a transition function `trans` that takes two arguments, the current state and the next input, and returns the next state. This new state is then passed to the `trans` function again, along with the next input. In other words, we can use `Fold[trans, initialState, listOfInputs]` to "turn the crank" of a finite state machine. For our purposes, a state is a list of collected but as-yet-undecoded bits, the initial state of the machine is an empty list, the inputs are the bits in the encoded message, and the state transition function can be defined as follows:

9. Versions of *Mathematica* earlier than 3.0 do not have the `Extract` function used here. Users of these versions can make the definition `Extract[list_, {indices__}] := list[[indices]]`.
10. The reason for this is simple: Since all characters are leaves in the code-generating tree, no character can be on the path to any other character.

```
trans[state_List, input_Integer] :=
With[{newstate = Append[state, input]},
    If[StringQ[code2char[tree, newstate]],
        decoded = decoded <>
                  code2char[tree, newstate];
        {},
    (* else *)
        newstate
    ]
]
```

Each time `trans` is called, the `input` (the next binary digit in the encoded list) is appended to `state` to form `newstate`. If `newstate` represents a complete code for a character, the corresponding character is appended to a string named `decoded` and `trans` returns an empty list as the new state. Otherwise, `trans` returns `newstate`.

To better understand the process, the following example uses `FoldList` to view the states that the finite state machine travels through while decoding `{0, 1, 0, 0, 1, 1, 1, 1, 1}`. `{0, 1}` is the code for `"s"`; `{0, 0, 1}` is the code for `"h"`; `{1, 1, 1}` is the code for `"e"`; and the final `1` is the beginning of an incomplete character code.

```
decoded = "";
FoldList[trans, {}, {0, 1, 0, 0, 1, 1, 1, 1, 1}]
{{}, {0}, {}, {0}, {0, 0}, {}, {1}, {1, 1}, {}, {1}}
```

The third element of this result (an empty list) is the state after the bits `{0, 1}` have been read. Since `{0, 1}` is a valid code (`code2char[{0, 1}]` ⇒ `"s"`), the new state is reset to the initial state, which is the empty list. Each of the succeeding empty lists has a similar explanation.

After the execution of the state machine the decoded text is contained in the global symbol `decoded`.

```
InputForm[decoded]
"she"
```

Here is the entire message decoded.

The value returned by `Fold` is the final state of the finite state machine.

```
decoded = "";
Fold[trans, {}, encodedText]
{}
```

The result of interest is in the global variable `decoded`.

```
InputForm[decoded]
"she sells sea shells by the sea shore"
```

This solution is not ideal because it uses a global variable (`decoded`) to accumulate the result. It would be "more functional" if the decoded message itself were returned by the call to `Fold`. We can accomplish this by changing the notion of a state to the form

`{string, list}`, where `string` is the so-far-decoded text and `list` is the list of collected but as-yet-undecoded bits.

```
Clear[trans]
trans[state_List, input_Integer] :=
With[{str = state[[1]],
      newpos = Append[state[[2]], input]},
    If[StringQ[code2char[tree, newpos]],
        {str <> code2char[tree, newpos], {}},
        {str, newpos}
    ]
]
```

The initial state will now be `{"", {}}`. Here's how it works:

Displaying the result in `InputForm` makes the empty strings explicit.

```
FoldList[trans, {"", {}},
         {0, 1, 0, 0, 1, 1, 1, 1, 1}] // InputForm
{{"", {}}, {"", {0}}, {"s", {}}, {"s", {0}},
  {"s", {0, 0}}, {"sh", {}}, {"sh", {1}},
  {"sh", {1, 1}}, {"she", {}}, {"she", {1}}}
```

Once again, the entire message is decoded.

```
Fold[trans, {"", {}}, encodedText]
{she sells sea shells by the sea shore, {}}
```

Of course, we have to extract the first element of the result for the final answer. The complete decoding routine is shown below:

The purpose of the `Block` will be explained below.

```
huffmanDecode[tr_List, input_List] :=
Block[{tree = tr},
    Fold[trans, {"", {}}, input][[1]]
]

huffmanDecode[tree, encodedText]
she sells sea shells by the sea shore
```

Note the use of the `Block` command (Section 4.5.1) to override the value of the global symbol `tree`, which is used by the `trans` function. This is an example of "something you should never do," but trying to write `trans` without using any global variables is not easy. We'll present an elegant solution to this problem next.

Embedding the code tree in the transition function

There are at least two solutions to the problem of `trans` using the global symbol `tree`. The first one is to add the code tree to the notion of a state. The state would then be of the form `{tree, string, list}`, and the initial state would be `{tree, "", {}}`. This is inelegant and inefficient, since the code tree (which could be a large expression) never changes throughout any particular call to `huffmanDecode`.

A better alternative is to construct a customized `trans` function for any given code tree. We can accomplish this by having `trans[tree]` return a pure-function state tran-

sition function that has the code tree embedded within it. Then the finite state machine can be implemented as `Nest[trans[tree], {"", {}}, listOfBits]`. This is both more elegant and more efficient than the previous suggestion, and as a bonus, it requires only a small change to the definition of `trans`:

```
Clear[trans]
trans[tr_List] :=
Function[{state, input},
    With[{str = state[[1]],
          newpos = Append[state[[2]], input]},
        If[StringQ[code2char[tr, newpos]],
            {str <> code2char[tr, newpos], {}},
            {str, newpos}
        ]
    ]
]
```

The actual state transition function used in the example is thus:

```
trans[tree]
Function[{state$, input$},
  With[{str$ = state$[[1]],
    newpos$ = Append[state$[[2]], input$]},
   If[StringQ[code2char[{{{{y, a}, h}, s},
       {{1, {{b, o}, {r, t}}}, { , e}}}, newpos$]],
    {str$<>code2char[{{{{y, a}, h}, s},
       {{1, {{b, o}, {r, t}}}, { , e}}}, newpos$],
     {}}, {str$, newpos$}]]]
```

Note that the entire code tree is textually embedded into the returned function. Also note that `trans[tree]` is evaluated only once, so the substitution is performed only once — in contrast to passing the entire tree along with the state, which would resubstitute the tree into the body of the `trans` function on every call to `trans`.

Finally, the `huffmanDecode` function needs to be updated to use the new `trans`:

```
Clear[huffmanDecode]
huffmanDecode[tr_List, input_List] :=
    Fold[trans[tr], {"", {}}, input][[1]]
```

Here's proof that `trans` no longer depends upon the global symbol `tree`.

```
someOtherTree = tree;
Clear[tree]
huffmanDecode[someOtherTree, encodedText]
she sells sea shells by the sea shore
```

Exercises

1. Write a function that packs a list of binary digits into a list of byte codes (8-bit integers). You can use the `Partition` function to break up the list of binary digits into sublists of length 8, and then map Horner's rule (Exercise 5.3.3.1) onto each sublist

to convert it to a byte code. For the ultimate in compactness you can use `FromCharacterCode` to convert the list of byte codes into a string.

Note that if the number of binary digits in the list is not an integral multiple of 8, you will lose the "leftovers" when the list is partitioned. Therefore, you will need to pad the list out to a length that is an integral multiple of 8 before partitioning. However, padding could introduce garbage into the message when it is decoded, so you also will need to store the number of bits with the packed representation so that it can be unpacked correctly.

2. Write a function that unpacks the byte-code representation that you created in the previous exercise. `IntegerDigits[int, 2]` can be used to convert an integer `int` into a list of binary digits. *Caveat:* Simply mapping `IntegerDigits[#, 2]&` onto the list of byte codes and flattening the result do not give the correct answer!
3. The current implementation of `trans` is inefficient: Every time `code2char` is called, the encoding tree is subscripted from the root down to the current position. Rather than use a list of subscripts to represent the current state of the finite state machine, we could use a subtree of the encoding tree. The initial state would be the entire tree. As bits are read, the subtree becomes smaller and smaller until it becomes a leaf; at that point, a character is recognized and the state is reset to the entire tree. Now only one level of subscripting is performed for each bit read. Implement a version of `trans` that uses this approach. Don't forget to make the corresponding changes to `huffmanDecode`.

5.4 Recursion

Divide and conquer is an algorithmic strategy whereby problems are solved by breaking them down into one or more smaller problems of the same type. This procedure continues until the problems being considered are so small that their solutions are trivial. The subsolutions are then combined to form the solution to the original problem.

Recursion is the act of a function calling itself. Programming languages that support recursion (a class that includes nearly all modern languages) make the implementation of divide-and-conquer algorithms straightforward. This is because the mechanism of recursion handles the combination of the subsolutions automatically.

5.4.1 The basics

Without a doubt, the most famous example of a recursive function is the factorial:

```
fact[n_Integer] := If[n == 0, 1, n fact[n - 1]]
Array[fact, 5]
{1, 2, 6, 24, 120}
```

As you can see, the `fact` function calls itself. There's nothing wrong with doing this, so long as "the buck stops" somewhere (in this case, when n becomes 0).

It is instructive to trace the evaluation of `fact` for a small argument.

```
Trace[fact[3]]
```

```
{fact[3], If[3 == 0, 1, 3 fact[3 - 1]],
  {3 == 0, False}, 3 fact[3 - 1],
  {{3 - 1, -1 + 3, 2}, fact[2],
   If[2 == 0, 1, 2 fact[2 - 1]], {2 == 0, False},
   2 fact[2 - 1], {{2 - 1, -1 + 2, 1}, fact[1],
    If[1 == 0, 1, 1 fact[1 - 1]], {1 == 0, False},
    1 fact[1 - 1], {{1 - 1, -1 + 1, 0}, fact[0],
     If[0 == 0, 1, 0 fact[0 - 1]], {0 == 0, True}, 1}\
     , 1 1, 1}, 2 1, 1 2, 2}, 3 2, 2 3, 6}
```

n == 0? No.
Call fact[2]
n == 0? No.
Call fact[1]
n == 0? No.
Call fact[0]
n == 0? Yes! Return 1.
1*1⇒1, 2*1⇒2, 3*2⇒6

To solve a problem recursively, you have to think about two cases:

1. For nontrivial cases, how do I reduce the problem to a smaller one (or several smaller ones)? In the `fact` example this is obvious.
2. What is the *base case?* There has to be a way to halt the recursion.

Note that if you forget the base case (or if your function ever calls itself without changing the arguments), the recursion will not terminate. In *Mathematica*, you'll get a "soft landing." Don't try this in other programming languages!

Lack of a base case prevents the recursion from terminating. The computation was aborted manually.

```
badfact[n_Integer] := n*badfact[n - 1]
badfact[5]
$RecursionLimit::reclim:
   Recursion depth of 256 exceeded.
$Aborted
```

5.4.2 Recursion on lists

The principal data structure in the functional programming paradigm is the list. Since lists have basically the same structure no matter what size they are, recursion is a natural choice for implementing many operations on lists. Here is a recursive function that reverses the elements in a list:

```
rev[s_List] :=
    If[ s=={}, {},
        Append[rev[Rest[s]], First[s]]
    ]

rev[{1, 2, 3}]
{3, 2, 1}
```

The strategy of `rev` is this:

1. If the list is empty (the base case), the result is another empty list. Otherwise ...
2. Form a sublist consisting of the original list minus the first element.

3. Reverse the sublist.
4. Append what used to be the first element to the end of the reversed sublist.

We can see what's going on by using the `Trace` function.

This shows all of the calls to `Append` that occur during the evaluation of `rev[{1, 2, 3}]`.

```
tr = Trace[rev[{1, 2, 3}], Append]
{Append[rev[Rest[{1, 2, 3}]], First[{1, 2, 3}]],
  {Append[rev[Rest[{2, 3}]], First[{2, 3}]],
   {Append[rev[Rest[{3}]], First[{3}]],
    Append[{}, 3], {3}}, Append[{3}, 2], {3, 2}},
  Append[{3, 2}, 1], {3, 2, 1}}
```

The trace shows that `rev[{1, 2, 3}]` evaluates to `Append[rev[Rest[{1, 2, 3}]], First[{1, 2, 3}]]`. Since the arguments to a function are evaluated before the function is evaluated, the first argument evaluates to `rev[{2, 3}]`, which in turn produces (after the evaluation of `First` and `Rest`) `Append[rev[{3}], 2]`. Likewise, `rev[{3}]` evaluates to `Append[rev[{}], 3]`, and `rev[{}]` evaluates to `{}` (the empty list). Now that the base case has been reached (i.e., there is no further recursive call), the entire process "unwinds." Since `rev[{}]` has evaluated to `{}`, `Append[rev[{}], 3]` evaluates to `Append[{}, 3]` ⇒ `{3}`, so this is the value of `rev[{3}]`. Now `Append[rev[{3}], 2]` ⇒ `Append[{3}, 2]` ⇒ `{3, 2}`, so this is the value of `rev[{2, 3}]`. Finally the `1` is appended to this, creating the final result, `{3, 2, 1}`.

The great thing about recursion is that all of this mess is kept track of by the normal function-call mechanism. Recursion allows us to reason about divide-and-conquer problems at a very high level, and the details take care of themselves.

As another simple example, here's a function that finds the minimum value in a list. Basically, the function looks at the first two elements in the list and drops the smaller of the two before recursing. When there is only one element left, it must be the minimum, so it is returned.

We use the `Which` statement (Section 4.2.3) to avoid a sequence of nested `If`'s.

```
minimum[s_List] :=
    Which[ s == {}, Infinity,
           Length[s] == 1, s[[1]],
           s[[1]] > s[[2]], minimum[Drop[s, {1}]],
           True, minimum[Drop[s, {2}]]
    ]

minimum[{3, 5, 2, 6, 4}]
2
```

The `s == {}` case is executed only if `minimum[{}]` is called directly by the user. If you are at all unclear about how `minimum` works, you should trace the example computation.

A more interesting example of recursion is the *mergesort* algorithm for sorting a list. The strategy of mergesort is:

1. Split the list into two more or less equally sized pieces.
2. Sort each piece recursively.
3. Merge the sorted pieces.

Mergesort reduces the problem of sorting a list, which is relatively difficult, to the problem of merging two sorted lists, which is considerably easier. The base case is sorting a list of 0 or 1 element, which is trivial. Assuming for the moment the existence of a suitable merge function, here is a mergesort function:

Switch was introduced in Section 4.2.2.

```
mergesort[s_List] :=
    Switch[ Length[s],
        0, {},
        1, s,
        _, With[{half = Quotient[Length[s], 2]},
               merge[mergesort[Take[s, half]],
                     mergesort[Drop[s, half]]]
        ]
    ]
```

The `merge` function is not difficult to write either, since by assumption both of its arguments are already sorted. If either list is empty, return the other list. If neither is empty, compare their first elements. Remove the smaller of the two, `merge` what's left, and prepend the element that was removed.

```
merge[a_List, b_List] :=
    Which[
        a == {}, b,
        b == {}, a,
        a[[1]] < b[[1]],
              Prepend[merge[Rest[a], b], a[[1]]],
        True, Prepend[merge[Rest[b], a], b[[1]]]
    ]

merge[{1, 3, 5, 6, 12}, {2, 5, 10}]
{1, 2, 3, 5, 5, 6, 10, 12}
```

Here is `mergesort` in action.

```
Table[Random[Integer, {1, 100}], {12}]
{60, 76, 54, 92, 54, 99, 92, 57, 8, 80, 97, 10}

mergesort[%]
{8, 10, 54, 54, 57, 60, 76, 80, 92, 92, 97, 99}
```

A trace of `mergesort`'s execution would be quite lengthy; you might want to trace a very small `mergesort` of, say, two elements in each list.

Exercises

1. Write a recursive function that adds `1` to every element of a list.
2. Write a recursive function that computes the length of a list.
3. Another approach to finding the minimum of a list is sometimes called a *tournament* algorithm. The basic idea is:
   (a) Split the list into two more or less equally sized pieces.
   (b) Find the minimum of each piece recursively.
   (c) Return the smaller of the two minima.
   Implement a tournament-style minimum function.
4. Write a recursive version of `Map`.

5.5 Manipulating Normal Expressions

Up to this point we have restricted our application of functional programming operations to lists. Now we take into consideration the structure of general *Mathematica* expressions — of which lists are merely a special case — and show how functional operations can be used on them.

Recall that expressions can be either atomic (symbols, numbers, strings) or nonatomic. The general structure of a nonatomic, or *normal*, expression is *`head`*`[`*`part1`*`, ..., `*`partn`*`]`, where the head and each of the parts are other expressions. We can use the `FullForm` function to explore the internal representation of various expressions. For example,

```
FullForm[a + b]
Plus[a, b]
```

A list is just a normal expression with the head `List`:

```
FullForm[{a, b}]
List[a, b]
```

Since lists are represented internally just as any other expressions are, and since there are many functions for operating on lists, you might suspect that you can use these functions on more general expressions as well. In fact, almost anything you can do to a list you also can do to an arbitrary expression, which is a very powerful capability. In the remainder of this section we will highlight this capability.

5.5.1 Subscripting expressions

One particular list operation that can be applied to any expression is subscripting. This concept is important enough that it deserves special treatment.

You may have already seen the use of subscripting to pick out the right-hand side of a rule:

The second element of a rule is its right-hand side.

```
(a -> b) [[2]]
b
```

This is because of the way rules are represented internally.

```
FullForm[a->b]
Rule[a, b]
```

If the head of this expression had been `List` rather than `Rule`, we would naturally have expected the second part to be `b`.

Here is a slightly more interesting example of subscripting:

```
(a/b) [[2, 1]]
b
```

The reason for this result is that the internal form of `a/b` is `Times[a, Power[b, -1]]`. In other words, part 2 of this expression is the expression `Power[b, -1]`, and part 1 of that expression is `b`. If this doesn't make sense to you, then try the following experiment:

Convert all heads in the expression to `List`.

```
(a/b) /. {Times->List, Power->List}
{a, {b, -1}}
```

This list has the same *structure* as the original expression, and quite obviously part `{2, 1}` is `b`.

Recall that list subscripting always starts with `1`. This is true of *all* normal expressions: The first argument to a function is always part `1`, and so on.

```
(a/b)[[1]]
a

(a/b)[[2]]
1
-
b
```

However, part `0` is defined to be the head of the expression:

```
(a/b)[[0]]
Times
```

Alternatively, you can use the built-in function `Head`.

```
Head[a/b]
Times
```

Note that if the head of an expression is a normal expression, you can subscript inside of the head also. For example,

Part 0 of this expression is `f[0, 1]`, and part 2 of that expression is 1.

```
Position[f[0, 1][x], 1]
{{0, 2}}
```

Subscripting does not work on atoms, however. Although atoms have heads (which can be accessed using either `Head[a]` or `a[[0]]`), they have no parts. Witness:

This output format is misleading.

```
FullForm[1/2]
Rational[1, 2]
```

Rationals have heads ...

```
(1/2)[[0]]
Rational
```

but no parts.

```
(1/2)[[1]]
Part::partd:
                                1
   Part specification (-)[[1]]
                                2
     is longer than depth of object.
(1/2)[[1]]
```

The contents of an atom are *raw data* that cannot be accessed using subscripting. Corresponding to every type of atom are special functions that allow you to access the data inside the atom. These functions are listed in Table 5-3.

Table 5-3 Special access functions for atoms of various types.

Type of Atom	Special Access Functions
`Integer`	`IntegerDigits`

`Real`	`RealDigits`
	`MantissaExponent`
	`$NumberBits`[a]

`Rational`	`Numerator` `Denominator`
`Complex`	`Re` `Im`
`String`	`Characters` `ToCharacterCode`

a. This function is undocumented in *The Mathematica Book*, although there is an on-line usage message for it. Its return value is machine-dependent.

As with lists, the `Position` function searches an expression for instances of some sub-expression and returns the sequence of subscripts necessary to extract that part of the expression. For example,

Note the double braces.

```
Position[a/b, b]
{{2, 1}}
```

The extra braces are necessary because the sub-expression might occur more than once.

```
Position[a/(1 + Log[a]), a]
{{1}, {2, 1, 2, 1}}
```

The form returned by `Position` can be passed to a variety of other functions that operate on expressions; this allows for "pinpoint" modification of expressions. We'll see an example of this in the next section.

Exercise

1. Try to extract the symbol `x` from each of the following expressions. Use `FullForm` if you need help, and use `Position` only to check your answers.

```
a + b*x

a + b*x[c]

a + b*x[[c]] (* don't worry about the error message *)

x/y

y/x (* not as easy as it looks *)

2^2^x

x[a, b][c]
```

5.5.2 Levels in expressions

Levels in arbitrary expressions are counted just as they are in lists, of course. However, there are some finer points about levels that we have neglected until now because they aren't very interesting when all of the heads in an expression are `List`. In particular, we need to discuss how heads are treated with regard to level specification.

You might have noticed by now that heads seem to be ignored by level specifications! For example, here are "all" of the subexpressions in the expression `a/b`:

Where are the symbols `Times` and `Power`?

```
Level[a/b, {0, Infinity}]
{a, b, -1, 1/b, a/b}
```

Every function that takes a level specification (see Table 5-1 on page 106) also has an option called `Heads`. The default value of this option for most of these functions is

`False`, which causes heads to be ignored.[11] We can, of course, override this behavior.

```
Level[a/b, {0, Infinity}, Heads->True]
```

$$\{\text{Times}, a, \text{Power}, b, -1, \frac{1}{b}, \frac{a}{b}\}$$

Recall that in Section 5.1.3 we defined the level of a subexpression within an expression to be the number of subscripts needed to extract that subexpression from the overall expression. This definition also is correct for heads; for example,

It takes two subscripts to reach the head `Power`.

```
Position[a/b, Power]
{{2, 0}}
```

Sure enough, the symbol `Power` is at level 2.

```
Level[a/b, {2}, Heads->True]
{Power, b, -1}
```

Here is a more interesting example in which the one of the heads is itself a normal expression.

All of the atoms in this expression are at level 2.

```
Level[f[0, 1][g[x, y]], {2}, Heads->True]

{f, 0, 1, g, x, y}
```

Note, however, that `TreeForm` does not print them all at the same depth. `TreeForm` makes no attempt to format compound heads.

```
TreeForm[f[0, 1][g[x, y]]]
f[0, 1][[        ]
        g[x, y]
```

5.5.3 Functional operations on expressions

Here are some examples of applying functional operations to arbitrary expressions. We'll begin with `Map`.

This wraps `f` around each argument to `g`.

```
Map[f, g[a, h[b]]]
g[f[a], f[h[b]]]
```

Level specifications work in the usual way.

```
Map[f, g[a, h[b]], {2}]
g[a, h[f[b]]]
```

`Heads->True` directs `Map` (and its ilk) to operate on the head of an expression as well as its parts.

```
Map[f, g[a, h[b]], Heads->True]
f[g][f[a], f[h[b]]]
```

The `Heads` options can be combined with a level specification.

```
Map[f, g[a, h[b]], {2}, Heads->True]
g[a, f[h][f[b]]]
```

11. One exception to this rule is `Position`, for which the default is `Heads->True`. Thus, `Position` will find a subexpression anywhere within an expression.

In contrast, `Operate` maps a function onto *only* the head.

```
Operate[f, g[a, b]]
f[g][a, b]
```

As an aside, applications requiring the use of functional operations on heads of expressions are fairly esoteric; it's very difficult to find an example that isn't contrived.

`Apply[f, expr]` simply changes the head of `expr` to `f`.

```
Apply[f, g[a, h[b]]]
f[a, h[b]]

Apply[f, g[a, h[b]], {1}]
g[a, f[b]]
```

Recall that in Section 5.5.1 we used the following technique to visualize the structure of an expression:

```
(a/b) /. {Times->List, Power->List}
{a, {b, -1}}
```

The problem with this technique is that it's hard to generalize; it requires that all of the heads in the expression be specified explicitly. What we really would like to do is change all of the heads in an expression to `List`, no matter what they are. Changing the head of an expression is exactly what `Apply` does; a level specification of `{0, Infinity}` will cause `Apply` to work on every subexpression within the expression:

```
Apply[List, a/b, {0, Infinity}]
{a, {b, -1}}
```

The technique is completely general.

```
Apply[List, g[a + b, h[c/d]], {0, Infinity}]
{{a, b}, {{c, {d, -1}}}}
```

`MapAt` (Section 5.1.4) allows functions to be targeted to particular points within an expression:

```
MapAt[f, a + b, {2}]
a + f[b]

MapAt[f, a / b, {{1}, {2, 1}}]
f[a]
----
f[b]
```

This is essential for those cases in which the built-in algebraic simplification commands refuse to do what you want them to do.

Getting *Mathematica* to simplify this expression can be an exercise in frustration.

```
expr = Sqrt[u - 1]/Sqrt[u^2 - 1];
```

```
Simplify[expr]
Sqrt[-1 + u]
-------------
          2
Sqrt[-1 + u ]
```

Even using PowerExpand doesn't help.

```
PowerExpand[expr]
Sqrt[-1 + u]
-------------
          2
Sqrt[-1 + u ]
```

Solution: First factor the radical in the denominator ...

```
MapAt[Factor, expr, Position[expr, u^2 - 1]]
      Sqrt[-1 + u]
-----------------------
Sqrt[(-1 + u) (1 + u)]
```

and then use PowerExpand.

```
PowerExpand[%]
      1
-------------
Sqrt[1 + u]
```

Note that the third parameter to `MapAt` is of the same form as the result produced by the `Position` function — thus they work well as a team. There are several other functions that accept a positional specification of this form, including `Insert`, `Delete`, and `ReplacePart`.

Exercise

1. Explain what is going on here.

```
Position[Sqrt[1 - u]/Sqrt[1 - u^2], Sqrt]
{}
```

5.5.4 Miscellaneous list operations

As mentioned at the beginning of this section, nearly any list operation can be applied to arbitrary expressions, as long as it makes sense to do so from a structural point of view. Here are some examples.

The only difference between this expression and the list `{a, b, c, d}` is its head (`Plus` versus `List`).

```
FullForm[a + b + c + d]
Plus[a, b, c, d]
```

Thus you can use almost any list operation on it.

```
Drop[%, 1]
b + c + d

Append[%, e]
b + c + d + e

Take[%, {2, 3}]
c + d
```

You can reorder the parts of any expression whose head does not have the attribute `Orderless`.[12]

```
Reverse[f[a, b, c]]
f[c, b, a]

RotateLeft[f[a, b, c]]
f[b, c, a]
```

Any two expressions can be `Join`ed if they have identical heads.

```
Join[a*b, c*d]
a b c d
```

Exercises

1. Given that

```
Reverse[a^b]
b^a
```

   then why doesn't `Reverse[a/b]` reciprocate the fraction?

2. Explain what is going on here. (*Hint:* Try using a list instead of a sum.)

```
Partition[a + b + c + d, 2, 1]
a + 2 b + 2 c + d
```

5.6 Additional Resources

5.6.1 Standard packages

Examples of functional programming can be found in many of the standard packages. In particular, `Statistics`DataManipulation` and `LinearAlgebra`MatrixManipulation` rely heavily on these techniques.

5.6.2 Publications

[Wagon 91] is notable for its many elegant applications of functional programming using *Mathematica*.

Readers interested in the theoretical aspects of functional programming may consult [Henderson 80].

12. Orderless (i.e., commutative) functions automatically sort their arguments. Thus `Reverse[Plus[a, b]]` ⇒ `Plus[b, a]` ⇒ `Plus[a, b]`.

5.7 Appendix: Lisp–*Mathematica* Dictionary

For the benefit of experienced Lisp programmers, Table 5-4 draws analogies between some common operations in Lisp and their closest *Mathematica* equivalents.

Table 5-4 Lisp–*Mathematica* Dictionary

Lisp	*Mathematica*	Comments/Special Input Forms
`(a b c ...)`	`{a, b, c, ...}`	a is not a function
`(f a b ...)`	`f[a, b, ...]`	f is a function
`apply`	`Apply`	`@@`
`catch`	`Catch`	
`cond`	`Which`	
`lambda`	`Function`	`#, &`
`let`	`With`	
`mapcar`	`Map`	`/@`
`quote`	`Unevaluated`	
`reduce`	`Fold`	
`setq`	`SetDelayed`	`:=`
`throw`	`Throw`	

Lisp programmers should note that lists in *Mathematica* are implemented using arrays, not linked lists. Thus, some algorithms that might be efficient in Lisp are extremely inefficient in *Mathematica*. We'll discuss performance issues in detail in Chapter 10, "Performance Tuning."

6

Rule-Based Programming

The programming paradigms that we have studied so far can all be categorized as *imperative programming*, in which it is the programmer's job to state, step by step, how to carry out the solution to the problem. *Rule-based programming* is fundamentally different from this. In the rule-based paradigm, the programmer simply writes down a set of rules that specify what transformations should be applied to any expression that is encountered during the course of solving the problem. The programmer need not specify the order in which these rules are to be executed; the underlying programming system figures that out.

Rule-based programming is a very natural way to implement mathematical computation, since symbolic mathematics essentially consists of applying transformation rules to expressions (e.g., differentiation rules, tables of integrals). *Mathematica*'s versatile pattern-matching capability, which we have only begun to explore, often makes rule-based programming the paradigm of choice for *Mathematica* programmers.

6.1 Patterns

6.1.1 What is a pattern?

A *pattern* is a *Mathematica* expression that represents an entire class of expressions. The simplest example of a pattern is a single `Blank`, `_`, which represents any expression. Another example is `_f`, which represents any expression having `f` as its head. We have already used patterns like these as formal parameters in function definitions. We'll explore this use in greater detail in Section 6.2.

Patterns also can be used by a variety of built-in functions to alter the structure of expressions. For example, a replacement rule can have a pattern on its left-hand side.

This rule squares every real number in an expression.

```
3 a + 4.5 b^2 /. x_Real -> x^2
```

$$3\,a + 20.25\,b^2$$

This rule squares every integer in an expression. Note the exponent of b.

```
3 a + 4.5 b^2 /. x_Integer -> x^2
```

$$9\,a + 4.5\,b^4$$

Be careful what you wish for — you just might get it!

```
3 a + 4.5 b^2 /. x_Symbol -> x^2
```

$$(\text{Plus}^2)[(\text{Times}^2)[3,\ a^2],\ (\text{Times}^2)[4.5,\ (\text{Power}^2)[b^2,\ 2]]]$$

The preceding example shows that a pattern can match any parts of an expression, even heads.

Patterns themselves are expressions, and almost any part of a pattern can be given a temporary name, which allows a rule to extract and manipulate the parts of an expression. These temporary names are called *pattern variables*. Here are three examples:

In this case the pattern variable x refers to any expression with head f; it matches f[a] and f[a, b].

```
(f[a] + g[b])/f[a, b] /. x_f -> x^2
```

$$\frac{f[a]^2 + g[b]}{f[a,\ b]^2}$$

In this case, however, the pattern matches only those expressions having head f and a single argument, which is referred to as x.

```
(f[a] + g[b])/f[a, b] /. f[x_] -> x^2
```

$$\frac{a^2 + g[b]}{f[a,\ b]}$$

On the other hand, this pattern matches only those expressions with head f that have exactly two arguments, referred to as x and y.

```
(f[a] + g[b])/f[a, b] /. f[x_, y_] -> (x + y)^2
```

$$\frac{f[a] + g[b]}{(a + b)^2}$$

Note that in none of the previous three cases was the subexpression g[b] affected, because it does not have the correct head.

Keep in mind that patterns match expressions based on the internal forms of those expressions. This can cause a great deal of confusion when you are trying to modify an expression whose internal form is quite different from what appears on the screen. For example:

```
expr = x / Exp[y]
```

$$\frac{x}{E^y}$$

This is bewildering ...

```
expr /. Exp[z_] -> z
-(x y)
```

until one examines the internal forms. The pattern changed the expression `E^(-y)` to `-y`.

```
FullForm /@ { x / Exp[y], Exp[z_] }
{Times[Power[E, Times[-1, y]], x],
  Power[E, Pattern[z, Blank[]]]}
```

6.1.2 Destructuring

A pattern can be constructed from any expression merely by substituting `Blanks` for various subexpressions. (The name `Blank` is meant to connote the idea, "Fill in the blank.") Furthermore, any `Blank` can be given a name and used as a pattern variable.

The pattern variable `x` matches the head of any expression having a single part.

```
f[a] + g[b] /. x_[_] -> x
f + g
```

Now `x` matches the head of any expression having exactly two parts: In this particular example, that is `Plus[a, b]`.

```
f[a] + g[b] /. x_[_, _] -> x
Plus
```

The technique of extracting parts of patterns using pattern variables is sometimes called *destructuring*. Destructuring combined with rule replacement is a very powerful capability.

This extreme example illustrates that it's possible to do just about anything with destructuring.

```
f[a] + g[b] /. x_[y_] -> y[x]
a[f] + b[g]
```

Practically speaking, destructuring and rule replacement are often easier than using `MapAt` (see Section 5.1.4) to modify a subexpression, and it is almost always easier to understand at a glance what a destructuring operation is doing, as opposed to what part of an expression is referred to by a long sequence of subscripts. The only time you would have to use subscripting instead of destructuring is when an expression contains multiple subexpressions with the exact same structure, only one of which you wish to extract or modify.

Suppose we have a list of `{x, y}` data points that we wish to plot as a logplot.[1] A functional way to transform the data would be as follows:

```
data = {{x1, y1}, {x2, y2}, {x3, y3}, {x4, y4}};
```

1. For educative purposes, let us agree to ignore the fact that the `Graphics`Graphics`` package contains a function to do logplots.

Separate the x and y values.

```
Transpose[data]
{{x1, x2, x3, x4}, {y1, y2, y3, y4}}
```

Use MapAt to transform the y values. This operation relies on the fact that Log is Listable (Section 3.3).

```
MapAt[Log, %, {2}]
{{x1, x2, x3, x4}, {Log[y1], Log[y2], Log[y3],
   Log[y4]}}
```

Recombine the x values and transformed y values.

```
Transpose[%]
{{x1, Log[y1]}, {x2, Log[y2]}, {x3, Log[y3]},
  {x4, Log[y4]}}
```

On the other hand, this type of data transformation is trivial to do by using pattern matching:

The y values are transformed in place.

```
data /. {x_, y_} -> {x, Log[y]}
{{x1, Log[y1]}, {x2, Log[y2]}, {x3, Log[y3]},
  {x4, Log[y4]}}
```

Exercise

1. What happens in the following case?

```
{{x1, y1}, {x2, y2}} /. {x_, y_} -> {x, Log[y]}
```

6.1.3 Testing patterns

There are two functions for testing patterns to see what kinds of expressions they will match: MatchQ and Cases.

The MatchQ predicate simply tests to see if a pattern matches an expression:

```
MatchQ[a + b + c, _Plus]
True
```

A more sophisticated function is Cases, which picks out all expressions in a list that match a given pattern. Cases is very useful for "debugging" your patterns.

If you don't understand why the pattern fails to match a+a, look at its FullForm.

```
Cases[{a, a + b, a + a}, x_ + y_]
{a + b}
```

The second argument to Cases can be a rule, in which case the rule is applied to each of the matched expressions before returning.

```
Cases[{a, a + b, a + a}, x_ + y_ -> y]
{b}
```

The head of the first argument to Cases need not be List. By default, Cases searches only at level 1.

The only integer at level 1 in `Times[5, (a + b)^6]` is 5.

```
Cases[5 (a + b)^6, _Integer]
{5}
```

`Cases` takes a level specification (Section 5.1.3) as an optional third argument, which tells it which levels to search for matches.

This finds integers at all levels.

```
Cases[5 (a + b)^6, _Integer, Infinity]
{5, 6}
```

By default, `Cases` does not search heads.

```
Cases[5 (a + b)^6, _Symbol, Infinity]
{a, b}
```

You can override this by setting `Heads->True`.

```
Cases[5 (a + b)^6, _Symbol, Infinity, Heads->True]
{Times, Power, Plus, a, b}
```

Exercises

1. Use `Select` and `MatchQ` to implement your own version of `Cases`. Don't worry about level specifications.
2. Why doesn't the pattern `x_ + y_` match the expression `a + a`?
3. Create a pattern that matches a symbol raised to an integer power. The pattern should match expressions such as `x^3` or `y^2`, but should not match `x^(2/3)` or `(x + y)^2`.
4. Does the pattern from the previous exercise match `x^1`? Why not? (*Note:* You don't yet have the tools to fix this problem, so don't attempt to do so.)

6.1.4 The role of attributes

Mathematica can be surprisingly intelligent about destructuring. Consider this example:

Even though this expression and this pattern have different structures ...

```
Length /@ {a + b + c, x_ + y_}
{3, 2}
```

the pattern nevertheless matches.

```
a + b + c /. x_ + y_ -> {x, y}
{a, b + c}
```

The reason that the pattern matched is that *Mathematica* knows that `Plus` is an associative operator, i.e., `a + b + c == a + (b + c)`. This knowledge is encoded into the attributes of the `Plus` function:

The `Flat` attribute means that `Plus` is associative.

```
Attributes[Plus]
{Flat, Listable, NumericFunction, OneIdentity,
  Orderless, Protected}
```

We've already seen `Listable`. `Flat` means that `Plus` is associative, as we saw above. `OneIdentity` means that `Plus[x] == x`. `Orderless` means that `Plus` is commuta-

tive, i.e., `Plus[a, b] == Plus[b, a]`. `Protected` means that you can't define new rules for `Plus` without first `Unprotecting` it.[2]

The significance of the `Orderless` attribute is that this pattern matches either of these expressions.

```
Cases[{a + b*c, a*b + c}, x_ + y_*z_]
{a + b c, a b + c}
```

This sort of behavior allows some fairly sophisticated transformations with a minimum of effort. For example, the following rule will expand a product of any number of factors, as long as any of them contain at least two additive terms:

```
expandrule = x_ (y_ + z_) -> x y + x z;
```

`y_ + z_` matches `b + c` and `x_` matches everything else, because `Times` is `Flat` and `Orderless`.

```
a (b + c) (d + e + f) /. expandrule
a b (d + e + f) + a c (d + e + f)

% /. expandrule
a b d + a c d + a b (e + f) + a c (e + f)

% /. expandrule
a b d + a c d + a b e + a c e + a b f + a c f
```

Rather than applying a rule such as this over and over, we can use `FixedPoint` to apply the rule to the expression repeatedly until the expression stops changing.

```
FixedPoint[# /. expandrule &, a (b + c) (d + e + f)]
a b d + a c d + a b e + a c e + a b f + a c f
```

This is a common enough kind of operation that a function exists expressly for this purpose. It is called `ReplaceRepeated`, and it has the special input form "`//.`":

```
a (b + c) (d + e + f) //. expandrule
a b d + a c d + a b e + a c e + a b f + a c f
```

6.1.5 Functions that use patterns

We have already used some functions that can take patterns as arguments, although we didn't know it at the time. The second argument to the `Count` and `Position` functions is actually a pattern. So, for example,

```
Count[{a, a + b, a + b + c, a + a}, x_ + y_]
2

Position[{a, a + b, a + b + c, a + a}, x_ + y_]
{{2}, {3}}
```

2. See Section 6.5, "Overriding Built-in Functions."

Like so many other functions that accept patterns, `Count` and `Position` can also be given level specifications to target their search.

There also is a function called `DeleteCases` that, as you might expect, returns the complement of what `Cases` returns. Like `Cases`, `DeleteCases` can operate on expressions having any head, and it also takes a level specification as an optional argument.

```
DeleteCases[{a, a + b, a + a}, x_ + y_]
{a, 2 a}

DeleteCases[5 (a + b)^6, _Integer]
(a + b)^6

DeleteCases[5 (a + b)^6, _Integer, {2}]
5 (a + b)
```

6.2 Rules and Functions

We'll see in this section that there is an intimate connection between rules and functions. Before proceeding, however, we need to introduce a new type of rule.

6.2.1 Delayed rules

Just as there are two types of assignment operations, `Set` and `SetDelayed`, so there are two types of rules: `Rule` and `RuleDelayed`. The difference between the two forms is that the right-hand side of a `RuleDelayed` is not evaluated until *after* the pattern variables have been substituted. Delayed rules can be specified using the syntax `a :> b`.

As an example of when you would need this behavior, consider once again the problem of trying to cancel `Sqrt[u - 1]` from the following expression:

```
expr = Sqrt[u - 1]/Sqrt[u^2 - 1];
```

We solved this problem in Section 5.5.3 by using `MapAt` to `Factor` the expression inside the radical. As an alternative, we might use the following rule:

All that remains to do is to apply `PowerExpand` to this result.

```
expr /. u^2 - 1 -> (u - 1) (u + 1)
      Sqrt[-1 + u]
-----------------------
Sqrt[(-1 + u) (1 + u)]
```

This rule is fine for this simple case, but in a more complicated case it would be a burden to have to supply the factored form of the polynomial. In that case we might try a rule of the following form:

```
expr /. 1/Sqrt[y_] -> 1/Sqrt[Factor[y]]
Sqrt[-1 + u]
------------
          2
Sqrt[-1 + u ]
```

The problem is that `Factor[y]` evaluates immediately to just plain y, and so the effect of the rule is to replace y_ with y — no change at all! Preventing the right-hand side of the rule from being evaluated before substitution is the solution to problems like this one:

In this case, `Factor[y]` is not evaluated until y has been replaced by the argument of `Sqrt`.

```
expr /. 1/Sqrt[y_] :> 1/Sqrt[Factor[y]]
      Sqrt[-1 + u]
-------------------------
Sqrt[(-1 + u) (1 + u)]
```

Finally, `PowerExpand` the result to effect cancellation.

```
PowerExpand[%]
     1
-----------
Sqrt[1 + u]
```

Exercises

1. Use a rule to numerically evaluate (i.e., apply the `N` function to) the `Log` term in the following expression, but *not* the `Sin` term. Compare the effects of using `Rule` and `RuleDelayed`.

```
Log[3] + Sin[3]
```

2. Why did we use `1/Sqrt[y_]` to match the denominator of the example expression in this section, i.e., why doesn't the simpler pattern `Sqrt[y_]` work?

6.2.2 Function definitions are rules

When you define a function in *Mathematica*, you actually are defining a rule. The rule(s) for a function *f* can be displayed using `DownValues[f]`:

```
fact[n_Integer] := Times @@ Range[n]
DownValues[fact]
{HoldPattern[fact[n_Integer]] :> Times @@ Range[n]}
```

As you can see, this definition has created a delayed rule that represents what should happen to expressions that match the pattern `HoldPattern[fact[n_Integer]]`. `HoldPattern` is a head that, like `Hold`, keeps its argument from evaluating.[3] Unlike `Hold`, however, `HoldPattern` is ignored for pattern-matching purposes. `HoldPattern` will be discussed in greater detail in Section 7.2.5.

3. Prior to version 3.0, `HoldPattern` was known as `Literal`. Users of those versions will see `Literal` everywhere this book shows `HoldPattern`.

The difference between a rule created by a function definition and an ordinary rule is that the former is globally applied, whereas the latter is applied only within a `Replace`-type operator (e.g., "`/.`"). Whenever the *Mathematica* kernel matches an expression to a global rule, it will replace it with the right-hand side of the rule. In fact, you can think of the way that the kernel evaluates an expression (in a somewhat oversimplified way) as:

```
expression //. { all global rules }
```

In other words, global rules continue to be applied *until the expression stops changing*. It is for this reason that the *Mathematica* kernel's main evaluation routine is sometimes called an *infinite evaluation* system. It also explains why it is so difficult to evaluate an expression only partway (see Section 7.3, "Working with Held Expressions").

According to Roman Maeder, one of the designers of the *Mathematica* programming language, pattern matching and rule substitution are "the underlying mechanism for implementing all other programming constructs" in *Mathematica* [Maeder 94a]. This remarkable fact, together with the uniform representation of everything as expressions, is what gives *Mathematica* its great power and flexibility.

6.2.3 Multiple definitions for the same symbol

Mathematica allows you to write multiple function definitions for the same symbol, to cover (presumably) different cases; these definitions become separate rules. Here is a recursive definition of `fact` in which we have defined the base case as a separate rule:

```
Clear[fact]
fact[0] = 1;
fact[n_Integer] := n*fact[n - 1]
```

There are now two rules for `fact`.

```
DownValues[fact]
{HoldPattern[fact[0]] :> 1,
  HoldPattern[fact[n_Integer]] :> n fact[n - 1]}
```

Having multiple definitions for the same function might seem analogous to *overloading* in languages such as C++ [Stroustrup 91], but in fact it is much more powerful. First of all, *Mathematica* patterns can test for much more than just the "type" of an argument; we'll discuss the full generality of patterns in Section 6.3.

Less obvious, but perhaps more important, is that the different alternatives do not have to be disjoint: Note that zero is an integer, yet when the argument to the `fact` function is zero, the rule for `fact[0]` is used rather than `fact[_Integer]`. This illustrates a general strategy of *Mathematica*'s pattern-matching engine: *It always tries to apply more specific rules before more general rules.* `DownValues[f]` displays the rules for `f` in the order in which the kernel tries to apply them. You can also see the order in which the rules for a symbol are applied by using `?sym`.

```
?fact
Global`fact

fact[0] = 1

fact[n_Integer] := n*fact[n - 1]
```

In cases where it is not obvious which rule is more specific, *Mathematica* keeps the rules in the order in which they were entered. In the `fact` example, the internal ordering would be the same no matter which rule was entered first, since the pattern `0` is more specific than the pattern `_Integer` (see Exercise 6.2.3.1, below). In other cases *Mathematica* won't always be able to figure out the correct order for the rules.[4] In such cases you can reorder the `DownValues` for a symbol explicitly. (There is an example of doing so in Section 6.5.3.)

Finally, note that we could have used either = (`Set`) or := (`SetDelayed`) to define the base case for `fact`, since its right-hand side is a constant.

Exercises

1. Clear the definition of `fact` and redefine it, but this time specify the rule for `fact[0]` last. Now examine the rule set for `fact`.
2. Implement a function that has the same functionality as `Take`, using a different rule for each possible form of the subscript specification.

6.2.4 Advantages of rule-based function definition

There are many advantages to implementing a function as a collection of rules rather than as one monolithic piece of code.

First, rule-based functions usually are faster than all-in-one function definitions, because in many cases *Mathematica*'s pattern-matching engine can evaluate alternatives faster than explicit conditional statements (e.g., `If`, `Switch`) can.

Second, using multiple rules to define functions is very natural for mathematicians. For example, here is a definition of the absolute value function that parallels the way the definition would be presented in a mathematics textbook:

```
absval[x_] := x     /; x >= 0
absval[x_] := -x    /; x < 0
```

(The "`/;`" construct will be discussed in Section 6.3.1, "Pattern constraints.")

---

4. Unlike logic programming languages such as Prolog [Clocksin & Mellish 84], *Mathematica* does not implement *backtracking*. This means that although the kernel tries to guess the best order in which to apply rules, if it turns out to have made a wrong choice, it makes no attempt to try some other order. Maeder develops a Prolog-style backtracking interpreter in *Mathematica* in [Maeder 94b].

Third, much of mathematics simply involves rewriting expressions according to recognized patterns (e.g., differentiation and integration). When you do mathematical simplification by hand, you are doing the pattern matching in your head. Thus, rule-based programming often produces clear, elegant solutions to mathematical problems. We will see examples of this throughout the remainder of the book.

Fourth, we don't need to limit our thinking to regarding global rules as functions in the algorithmic sense. The ability to create arbitrarily many definitions for `f[`*`constant`*`]` allows us to think of global rules as a way of storing knowledge — a *rule base*. The standard packages `Miscellaneous`CityData`` and `Miscellaneous`ChemicalElements`` are excellent examples of this way of thinking about global rules.

Here is an example of the kind of knowledge contained in the `ChemicalElements` package.

```
Needs["Miscellaneous`ChemicalElements`"]
Through[{AtomicWeight, MeltingPoint, BoilingPoint,
        HeatOfFusion, Density,
        ThermalConductivity}[Cesium]]
```

$$\{132.9054,\ 301.55\ \text{Kelvin},\ 951.6\ \text{Kelvin},\ \frac{2.09\ \text{Joule}}{\text{Kilo Mole}},\ \frac{1873.\ \text{Kilogram}}{\text{Meter}^3},\ \frac{35.9\ \text{Watt}}{\text{Kelvin Meter}}\}$$

Fifth, a *Mathematica* function can augment its own rule base as it executes, allowing time-consuming computations to be *cached*. We will explore this idea in greater detail in Section 6.4, "Dynamic Programming."

6.2.5 Clearing definitions selectively

Sometimes when you are developing a new function, you make a mistake and need to redefine the function. Rather than doing a `Clear[f]`, which requires starting all over, you can selectively clear a single definition using "`f[`*`patt`*`] =.`". This operation is called `Unset`.

Here's a botched attempt to extend the definition of factorial to the real numbers (`fact[n_]` should evaluate to `Gamma[n + 1]`).

```
fact[n_] := Gamma[n]

fact /@ {3.99, 4, 4.01}
{5.92519, 24, 6.07593}
```

We attempt to redefine the bad rule, but it seems to have no effect.

```
fact[m_] := Gamma[m + 1]

fact /@ {3.99, 4, 4.01}
{5.92519, 24, 6.07593}
```

The reason is that the old rule for `fact[n_]` is still there, and it comes before the new rule for `fact[m_]`.

```
?fact
Global`fact

fact[0] = 1

fact[n_Integer] := Apply[Times, Range[n]]
```

```
fact[n_] := Gamma[n]

fact[m_] := Gamma[m + 1]
```

As you can see, *Mathematica* considers two structurally identical patterns to be different if the names of the pattern variables are not identical! This can result in extremely perplexing behavior. `Unset` is the "magic bullet" that can fix this situation:

This clears the bad rule without touching any of the others.

```
fact[n_] =.
```

Now the `fact` function works as intended.

```
fact /@ {3.99, 4, 4.01}
{23.6415, 24, 24.3645}
```

If you ever botch something so badly that you can't seem to get the rules straight, and reentering them all would be unacceptable, you can always directly modify the function's `DownValues` by assigning to `DownValues[f]`, e.g.,

```
DownValues[f] := Drop[DownValues[f], {i}]
```

where `i` is the subscript of the rule you want to get rid of.

6.2.6 "Pure" rule-based programming

Rather than writing rules as function definitions, you can apply the rules to expressions locally using `ReplaceRepeated`, essentially mimicking the operation of the kernel. For example, here's an alternative way to compute the factorial. Note that you *must* specify the rules in the order shown, or else the rule for `f[0]` will never be used!

`ReplaceRepeated` continues to apply rules from the given rule set until the expression stops changing.

```
f[5] //. {f[0] :> 1, f[n_] :> n*f[n - 1]}
120
```

In contrast to recursion, this technique does not build up a large evaluation stack, and hence is not constrained by the `$RecursionLimit` "safety net."

```
Table[Depth[Trace[fact[i]]], {i, 10}]
{6, 7, 8, 9, 10, 11, 12, 13, 14, 15}

Table[Depth[Trace[
        f[i] //. {f[0] :> 1, f[n_] :> n*f[n - 1]}]],
      {i, 10}]
{9, 9, 9, 9, 9, 9, 9, 9, 9, 9}
```

This may be seen as an advantage or a disadvantage, depending on the circumstances.

`ReplaceRepeated` and its relatives scan lists of rules sequentially for pattern matches. For long lists of rules, `Replace`-type operations may be sped up by using the

`Dispatch` function to generate a *dispatch table* for the list of rules. See §2.4.2 of *The Mathematica Book* for details.

6.3 Pattern Building Blocks

Mathematica provides a rich set of building blocks for constructing patterns. We cannot do more than provide a few examples of each here. You will come to really appreciate what patterns can do as you read the remainder of this book and other *Mathematica* literature.

6.3.1 Pattern constraints

There are three constructs that can be used to constrain patterns. Two of them constrain the values of individual `Blanks`, while the third one can be used to specify relationships between different pattern variables.

_head

As we saw in Section 4.1.3, "Type checking," a `Blank` can be constrained to match only those expressions with a particular head by appending the head to the `Blank`.

This pattern matches only integers.

```
Cases[{1, Sqrt[2], b}, _Integer]
{1}
```

_?test

A `Blank` also can be constrained using the form `_?test`, where *`test`* can be any function of a single argument. If the application of *`test`* to the expression that matches the `Blank` returns `True`, then the pattern matches.

Here's another way to write a pattern that matches only integers.

```
Cases[{1, Sqrt[2], b}, _?IntegerQ]
{1}
```

Note that, given the choice, it's more efficient to match the head *structurally* using `_Integer` than to use `_?IntegerQ`, so you probably wouldn't use a test function in this simple case. However, many predicates exist (see Section 3.4) that can test for more complicated conditions, such as whether or not an expression is an atom (`AtomQ`) or a number (`NumberQ`). And in addition to using the system-defined predicates, you can of course define your own.

Here's a pattern that matches an expression only if it is a number between 1 and 10.

```
between[x_] := 1 <= x <= 10
Cases[Table[Random[Real, {1, 100}], {20}], _?between]
{7.91991, 9.15871}
```

You also can combine head matching with a test function.

This pattern matches only *nonnegative* integers.

```
Cases[{-1, 0, 1, 1.5}, _Integer?NonNegative]
{0, 1}
```

pattern /; condition

A *condition* consisting of "/;" followed by any expression involving pattern variables may be attached to almost any part of a pattern. For example, any of the following rules could be used to define the recursive case of a `factorial` function:

```
factorial[n_Integer?NonNegative] := body
factorial[n_Integer /; NonNegative[n]] := body
factorial[n_Integer] /; NonNegative[n] := body
factorial[n_Integer] := body /; NonNegative[n]
factorial[n_] := body /; IntegerQ[n] && NonNegative[n]
```

Conditions are handy because they obviate the need to write a separate test function, e.g., the function `between` from the previous example.

This function accepts arguments that are between 1 and 10 only.

```
f[n_ /; 1 <= n <= 10] := ...
```

Conditions also are more flexible than the `?test` construct because they can involve multiple pattern variables.

This function will be called only if the first argument is less than the second.

```
f[x_, y_] /; x < y := ...
```

Note that in this particular case, you *must not* place the condition within the square brackets (e.g., `f[x_, y_ /; x < y]`); if you do so, *Mathematica* will assume that you want to test the pattern variable `y` against a *global* symbol called `x`, which is probably *not* what you had intended.

Thus, the only constraint on a condition is that if it involves multiple pattern variables, then it should appear outside of the smallest expression containing all of those variables. For both readability and efficiency reasons it's a good idea to place a condition as near as possible to the variable(s) it involves. The author prefers using the first of the four condition forms shown in the `factorial` example whenever possible. However, some people prefer the third form because it is reminiscent of the way functions are defined in mathematics books (cf. the definition of the `absval` function on page 150).

Exercises

1. Redefine `f` as follows:

```
Clear[f, x, y]
f[x_, y_ /; x < y] := True
f[_, _] := False
```

Now evaluate `f` using various arguments. What happens? Finally, evaluate the following:

```
x = 2;
f[99, 3]
```

2. Modify the `minimum` function from Section 5.4.2 so that it will not evaluate unless its argument is a list of numbers. (*Hint:* Use `ListQ` and `NumberQ`.)
3. Write a function to compute binomial coefficients called `binomial[n, r]` that evaluates only if `n` is greater than or equal to `r` and `r` is greater than or equal to `0`.

6.3.2 Patterns with default values

A default value can be defined for a pattern by using the *`pattern:value`* construct. This is a useful technique for providing optional arguments to functions.

This function uses Horner's rule (Section 5.3.3) to convert base-b digits into a base-10 integer.

```
toInteger[digits_List, b_:2] :=
    Fold[b #1 + #2 &, 0, digits]
```

By default, the digits are treated as base 2.

```
toInteger[{1, 0, 0, 1}]
9
```

Other bases can be used by supplying a second argument.

```
toInteger[{1, 0, 0, 1}, 3]
28
```

Note that when there is more than one pattern variable with a default value and there are not enough parts in the expression to fill them all, *Mathematica* will fill them in left-to-right order. Using the `Cases` function with a rule as the second argument, we can see what is being assigned to a pattern variable under different circumstances:

`b` gets a value before `c` does.

```
Clear[f, a, b, c]
Cases[{f[x, y, z], f[x, y], f[x]},
        f[a_, b_:2, c_:3] -> {a, b, c}]
{{x, y, z}, {x, y, 3}, {x, 2, 3}}
```

Some functions have predefined default values for certain arguments. These predefined defaults can be specified with a pattern of the form "`_.`". For example, the default value for an argument to the `Plus` function is `0`. Therefore, the pattern "`a_ + b_.`" matches any of the following:

```
Cases[{x, x + y}, a_ + b_. -> {a, b}]
{{x, 0}, {y, x}}
```

You can see what the default values for a function are by evaluating `DefaultValues[`*`name`*`]`:

```
DefaultValues[Times]
{HoldPattern[Default[Times]] :> 1}
```

The preceding explains why this pattern matches both of the given cases.

```
Cases[{x, x * y}, a_ * b_. -> {a, b}]
{{x, 1}, {y, x}}
```

Exercises

1. Modify your binomial coefficient function from Exercise 6.3.1.3 so that if `r` is omitted, the function returns `n`.
2. Clear `toInteger` and redefine it so that it checks that each of the digits in its first argument is a valid base-`b` digit. *Hint:* Write a condition clause that uses `VectorQ` and an appropriate pure function.
3. What values are assigned to the pattern variables `a`, `x`, and `b` by the pattern "`a_. * x_ + b_.`" when it is applied to expressions such as `x`, `x + y`, `w * x`, `w * (x + y)`, `w * x + y`, and `w * x + z * y`?
4. Examine the default values for the `Power` function. Use this information to write a pattern that matches both `x^y` and `x`.

6.3.3 Example: Writing a derivative function

Although *Mathematica* has a built-in differentiation function, writing one is a good way to illustrate the use of destructuring, conditions, and default values, and also to show how rule-based programming allows easy incremental enhancements to a function.

We will make heavy use of the `FreeQ` predicate. `FreeQ[`*`expr`*`, `*`form`*`]` returns `True` if no subexpression in *`expr`* matches *`form`* (which can be a pattern). Mathematically speaking, `FreeQ[`*`expr`*`, `*`var`*`]` means that *`expr`* does not depend on the variable *`var`*.

Here are the basic rules for differentiation. The first two embody the fact that differentiation is a linear operator:

```
diff[c_ * f_, x_] /; FreeQ[c, x] := c*diff[f, x]
diff[f_ + g_, x_] := diff[f, x] + diff[g, x]
```

Using no more than these two rules, `diff` can rewrite derivatives of linear expressions in terms of the derivatives of the parts that involve the variable `x`.

```
diff[3 x^2 - 2 x + 1, x]
```

$$\texttt{diff[1, x] - 2 diff[x, x] + 3 diff[x}^2\texttt{, x]}$$

We continue to add rules for `diff`, with each new rule building upon and enhancing the functionality of the previous ones. The derivative of a constant is 0:

```
diff[c_, x_] /; FreeQ[c, x] := 0
```

That got rid of the constant term in the previous result.

```
diff[3 x^2 - 2 x + 1, x]

-2 diff[x, x] + 3 diff[x^2, x]
```

The derivative of x^n is nx^{n-1}. Note that the use of "n_." for the exponent means that the first parameter also matches the expression x:

```
diff[x_^n_., x_] /; FreeQ[n, x] := n x^(n - 1)
```

So far so good ...

```
diff[3 x^2 - 2 x + 1, x]

-2 + 6 x
```

but `diff` doesn't work on expressions of this form.

```
diff[(x + 1)^2, x]

diff[(1 + x)^2, x]
```

We need to implement the *chain rule*. Below, `fx` is a mnemonic for "function of `x`."

```
diff[fx_^n_, x_] /; FreeQ[n, x] && !FreeQ[fx, x] :=
    n * fx^(n - 1) * diff[fx, x]

diff[(x + 1)^2, x]

2 (1 + x)

diff[(x^2 + 2 x + 1)^3, x]

3 (2 + 2 x) (1 + 2 x + x^2)^2
```

Chain rule seems to eliminate the need for the power rule, so we might try to `UnSet` the latter:

Note that to `UnSet` a rule, the left-hand side must be typed verbatim, *including conditions.*

```
diff[x_^n_., x_] /; FreeQ[n, x] =.

?diff

Global`diff

diff[(c_)*(f_), x_] /; FreeQ[c, x] := c*diff[f, x]

diff[(f_) + (g_), x_] := diff[f, x] + diff[g, x]

diff[c_, x_] /; FreeQ[c, x] := 0

diff[(fx_)^(n_), x_] /;
   FreeQ[n, x] && !FreeQ[fx, x] :=
  n*fx^(n - 1)*diff[fx, x]
```

Unfortunately, our rules now leave out one important case: `diff[x, x]`.

```
diff[(x^2 + 2 x + 1)^3, x]

3 (1 + 2 x + x^2)^2 (2 diff[x, x] + 2 x diff[x, x])
```

This is easily fixed by the following simple rule:

Note that this rule fires only if the two parameters are identical.

```
diff[x_, x_] := 1

diff[(x^2 + 2 x + 1)^3, x]
```

$$3\ (2 + 2\ x)\ (1 + 2\ x + x^2)^2$$

The beauty of the rule-based approach is that the functionality of `diff` can be extended at any time by adding new rules, without (hopefully) having to modify the existing rules. Some examples will be explored in the exercises.

Exercises

1. To fix the lack of a rule for `diff[x, x]`, why didn't we just modify the first parameter for the chain rule so that its exponent was of the form "n_."?
2. Add *product rule* to `diff`: $\frac{d}{dx}f(x)g(x) = f(x)\frac{d}{dx}g(x) + g(x)\frac{d}{dx}f(x)$.
3. Do you need a separate rule to implement the *quotient rule*?
4. Add rules for exponentials and logarithms to `diff`.
5. Add rules for trigonometric functions to `diff`.

6.3.4 Patterns that match more than one expression

There are two patterns that can be used to match sequences of expressions: __ (a double `Blank`, called `BlankSequence`) and ___ (triple `Blank`, called `BlankNullSequence`). The double `Blank` matches a sequence (comma-separated) of one or more expressions, and the triple `Blank` matches a sequence of zero or more expressions. (A sequence of zero length is equivalent to `Null`.)

The triple `Blank` matches everything the double `Blank` does, plus `Null`.

```
Clear[f]
Cases[{f[], f[1], f[1, 2]}, f[__]]
Cases[{f[], f[1], f[1, 2]}, f[___]]
{f[1], f[1, 2]}

{f[], f[1], f[1, 2]}
```

When qualified with a head test or predicate test, all of the expressions in a sequence must satisfy the test in order for the pattern to match.

`__Integer` matches a sequence of one or more integers, but not a mixture of integers and other types.

```
Cases[{f[1, 2, 3], f[1, a, 3]}, f[__Integer]]
{f[1, 2, 3]}
```

Probably the most common use for `Blank` sequences is destructuring lists of unknown lengths. This capability makes functions and rules that operate on lists much more compact and easy to write. For example, here's a reimplementation of the `rev` function of Section 5.4.2 that uses destructuring:

Note that a sequence is *not* a list; we must enclose r in list braces before we make the recursive call.

```
Clear[rev]
rev[{f_, r___}] :=  Append[rev[{r}], f]
rev[{}] = {};

rev[{1, 2, 3}]
{3, 2, 1}
```

We could also have written it like this.

```
Clear[rev]
rev[{f___, l_}] := Prepend[rev[{f}], l]
```

And here is the `minimum` function from Section 5.4.2 rewritten to use destructuring.

The comparison for the running minimum is now implemented by pattern matching. Also note that the list argument passed to the recursive call is "restructured" from the constituents of the original list.

```
Clear[minimum]
minimum[{}] = {};
minimum[{a_}] = {a};
minimum[{a_, b_, c___} /; a <= b] := minimum[{a, c}]
minimum[{a_, b_, c___}] := minimum[{b, c}]

minimum[{3, 5, 2, 6, 4}]
{2}
```

One of the more elegant examples of what can be accomplished by pattern matching and destructuring is the following function for sorting the elements of a list:

The second rule is a "catch-all" that fires only after the first rule no longer applies.

```
sorter[{a___, b_, c_, d___} /; b > c] :=
    sorter[{a, c, b, d}]
sorter[x_] := x

sorter[{3, 5, 2, 6, 4}]
{2, 3, 4, 5, 6}
```

`sorter` works because *Mathematica* tries to match the pattern variables b and c to every consecutive pair of elements in the input. Each time a pair of elements is swapped, pattern matching begins again at the left end of the list. Note, however, that this is quite inefficient: The number of comparisons increases as the square of the number of elements. By printing the pattern variables as a side effect of testing the condition, we can see all of the pattern-matching attempts, even those that fail.[5]

```
Clear[sorter]
sorter[{a___, b_, c_, d___}] := sorter[{a, c, b, d}] /;
    (Print[{a}, {b}, {c}, {d}, b > c]; b > c)
sorter[x_] := x
```

Students of computer science will recognize this as *bubblesort*, a classic example of how *not* to sort.

```
sorter[{3, 5, 2, 6, 4}]
{}, {3}, {5}, {2, 6, 4}, False
{3}, {5}, {2}, {6, 4}, True
{}, {3}, {2}, {5, 6, 4}, True
{}, {2}, {3}, {5, 6, 4}, False
{2}, {3}, {5}, {6, 4}, False
```

5. Alternatively, we could trace the computation, but the trace is harder to read.

```
{{2, 3}, {5}, {6}, {4}, False}
{{2, 3, 5}, {6}, {4}, {}, True}
{{}, {2}, {3}, {5, 4, 6}, False}
{{2}, {3}, {5}, {4, 6}, False}
{{2, 3}, {5}, {4}, {6}, True}
{{}, {2}, {3}, {4, 5, 6}, False}
{{2}, {3}, {4}, {5, 6}, False}
{{2, 3}, {4}, {5}, {6}, False}
{{2, 3, 4}, {5}, {6}, {}, False}
{2, 3, 4, 5, 6}
```

Another use for sequence-matching patterns is to implement functions that take a variable number of arguments. For example, in Section 5.1.1 we saw how to use `Apply` to define a function that computes the arithmetic mean of a list. Here is a function that computes the arithmetic mean of a sequence, rather than a list, of numbers:

```
amean[args__] := Plus[args]/Length[{args}]
amean[1, 2, 3, 4, 5]
3
```

There are two noteworthy techniques in this example. First, the sequence of arguments can be directly "plugged into" the head of your choice (in this case, `Plus`), and second, the number of expressions in a sequence can be found by wrapping it in list braces and passing the resulting list to `Length`. Furthermore, by using destructuring it is trivial to extend this version of `amean` to work on lists as well.

```
amean[{args__}] := amean[args]
amean[{1, 2, 3}]
3
```

Exercises

1. Use sequence-matching patterns to write one-liners that mimic the functionality of (a) `First`, (b) `Last`, (c) `Rest`, (d) `Apply`.
2. Use sequence matching to write recursive functions that mimic the functionality of (a) `Map`, (b) `Nest`, (c) `Fold`.
3. Redo Exercise 2 using the "pure" rule-based approach of Section 6.2.6.
4. Rewrite the `mergesort` function from Section 5.4.2 using the techniques of this section. Compare its performance to `sorter` for a wide range of list sizes, and graph the results.
5. Write a recursive function that creates a *perfect shuffle* of two lists, i.e.,

```
shuffle[{1, 2, 3}, {x, y, z}]
{1, x, 2, y, 3, z}
```

Be careful to handle cases in which the two lists have different lengths!

6.3.5 Application: Functions with options

A very common reason to have a variable number of arguments in a *Mathematica* function is to implement *options*. Options are named arguments of the form *name*`->`*value* or *name*`:>`*value* that follow the positional arguments in a function call.

The predicate `OptionQ` is available to test an expression to see if it is an option or a list of options. Since there may be zero or more options in a function call, the pattern `___?OptionQ` is exactly the right tool for the job. For example, here is how one could declare a function that has one positional parameter and an arbitrary number of options:

```
f[arg1_, opts___?OptionQ] := ...
```

Here are the kinds of expressions that are matched by the pattern variable `opts`:

```
Cases[{f[x], f[x, a->b], f[x, a:>b, c->d],
       f[x, {a->b, c->d}, e->f]},
      f[arg1_, opts___?OptionQ]->{opts}]
{{}, {a -> b}, {a :> b, c -> d},
  {{a -> b, c -> d}, e -> f}}
```

The use of rules for specifying options makes it quite easy to extract the values carried by the options. The sequence of options is wrapped in a list and then used on the right-hand side of a `ReplaceAll` operator ("`/.`"). However, a bit of care must be exercised because of the possible presence of nested lists in `{opts}`. For example,

The last value of `{opts}` in the previous example can't be used directly.

```
{a, c, e} /. Last[%]
ReplaceAll::rmix:
   Elements of {{a -> b, c -> d}, e -> f}
     are a mixture of lists and non-lists.
{a, c, e} /. {{a -> b, c -> d}, e -> f}
```

Simply `Flatten[{opts}]` before passing it to `ReplaceAll`. The values of the options are extracted.

```
{a, c, e} /. Flatten[Last[%%]]
{b, d, f}
```

Another possible fly in the ointment arises in the case of missing options — options are, after all, optional! To handle this possibility, simply follow the initial replacement with another set of options that carry the default values for each possible optional parameter. For example,

The user specifies only one of two possible options.

```
{name1, name2} /. {name1->value1}
{value1, name2}
```

The default value for the unspecified option is substituted afterward.

```
% /. {name1->default1, name2->default2}
{value1, default2}
```

The two ReplaceAll operations can be combined into one input because "/." is left-associative.

```
{name1, name2} /. {name1->value1} /.
    {name1->default1, name2->default2}
{value1, default2}
```

The techniques just illustrated work for any number of options, no matter how few or how many are supplied by the caller of the function.

Here is another version of the `toInteger` function introduced in Section 6.3.2. This version uses an option called `Base`, rather than a positional parameter, to specify the number base.

```
toInteger[digits_List, opts___?OptionQ] :=
With[{b = Base /. Flatten[{opts}] /. {Base->2}},
    Fold[b #1 + #2 &, 0, digits]
]
```

The base defaults to 2.

```
toInteger[{1, 0, 0, 1}]
9
```

A different base can be specified using `Base->`*b*.

```
toInteger[{1, 0, 0, 1}, Base->3]
28
```

By convention, default options for a function `f` can be inspected by using `Options[f]` and modified by using `SetOptions[f, `*option1*`, `*option2*`, ...]`. This and other fine points of option handling will be discussed in Section 9.2, "Options."

Exercise

1. Add an option called `Weights` to the `amean` function given at the end of Section 6.3.4. The function should behave as follows:

```
amean[{a, b, c}, Weights->{1, 2, 3}]
a + 2 b + 3 c
-------------
      6
```

The default value for `Weights` should be a list of all `1`s.

```
amean[{a, b, c, d}]
a + b + c + d
-------------
      4
```

6.3.6 Assigning names to entire patterns

The construct *n*`:`*p* assigns the name *n* to the entire pattern *p*. What makes this different from a construct such as *n*`_` is that *p* can be an arbitrarily complicated pattern. For example, the pattern `n:{___Integer}` matches the same class of expressions as `n_ /; VectorQ[n, IntegerQ]` or `n_?(VectorQ[#, IntegerQ]&)` — namely, a list of zero or more integers — but is faster *and* easier to understand.

We will illustrate the use of this construct by reimplementing the `merge` function from Section 5.4.2.

This rule fires if `a1` ≤ `b1`. Note the use of the pattern variable `b` to represent the entire second argument.

```
merge[{a1_, arest___}, b:{b1_, ___}] /; a1 <= b1 :=
    Prepend[merge[{arest}, b], a1]
```

This rule handles the symmetric case. Note the use of the pattern variable `a` to represent the entire first argument.

```
merge[a:{__}, {b1_, brest___}] :=
    Prepend[merge[a, {brest}], b1]
```

Finally, there are two base cases.

```
merge[a_List, {}] := a
merge[{}, b_List] := b
```

No comparison for `a1 > b1` is necessary in the second rule because that rule isn't checked until after the first rule fails to match. At that point, `a1 > b1` is the only possibility. That being the case, however, the astute reader may be wondering why the simpler pattern `merge[a_, {b1_, brest___}]` wasn't used instead. The answer will be revealed in the exercises.

Exercises

1. Compare the speed of the following implementation of `merge` with the one given in this section:

```
Clear[merge]
merge[{a1_, arest___}, b:{b1_, ___}] /; a1 <= b1 :=
    Prepend[merge[{arest}, b], a1]
merge[a_, {b1_, brest___}] :=
    Prepend[merge[a, {brest}], b1]
merge[a_List, {}] := a
merge[{}, b_List] := b
```

   Do you understand why this implementation is slower than the first one? *Hint:* Run the following benchmark under each implementation:

```
$RecursionLimit = 600;
Table[Timing[merge[{}, Range[i]];], {i, 100, 500, 100}]
```

2. Rewrite `merge` to use `ReplaceRepeated` instead of recursion.

6.3.7 Repeated patterns

The construct "*p*`..`" indicates one or more repetitions of the pattern *p*. The construct "*p*`...`" indicates 0 or more repetitions of *p*. We'll illustrate the use of the former with a function that performs a form of data compression known as *run-length encoding*.

Run-length encoding is a method for compressing data that has long runs of identical values — a bitmap image is a good example. A string of n repetitions of a number x is turned into `{x, n}`. For example, the run-length encoding of the data `{1, 0, 0, 1, 1, 1, 2, 2, 1}` would be

```
runEncode[{1, 0, 0, 1, 1, 1, 2, 2, 1}]
{{1, 1}, {0, 2}, {1, 3}, {2, 2}, {1, 1}}
```

Here is Stephen Wolfram's solution to the run-length encoding problem ([Wolfram 91] page 13), which exploits repeated patterns:

The pattern named `same` counts the number of consecutive `x`'s.

```
Clear[runEncode]
runEncode[{rest___Integer, same:(x_Integer)..}] :=
    Append[runEncode[{rest}], {x, Length[{same}]}]
runEncode[{}] = {};
```

Exercise

1. What is the difference between the patterns "`(x_Integer)..`" and "`x__Integer`"? Is there any difference between "`(_Integer)..`" and "`__Integer`"?

6.3.8 Alternative patterns

You can specify a pattern that matches one or more alternatives by separating the alternatives with the vertical bar "`|`":

Here's a pattern that matches either `a` or `b`.

```
{a, b, c} /. a | b -> d
{d, d, c}
```

If the rule's right-hand side contains a pattern variable, it must appear on both sides of the vertical bar.

```
{Log[2], 4.5, 2 + 3 I} /. x_Integer | x_Real -> Sqrt[x]
{Log[2]/2, 2.12132, 2 + 3 I}
```

You can see that if the same pattern variable did not occur in both alternatives in the second example, there would be no way to write the right-hand side of the rule correctly.

Alternatives and repeated patterns can be combined to write a pattern that matches a sequence of any number of elements, each of which matches one of a set of alternatives. For example, here is how one could write a pattern that matches a list containing only integers or symbols:

```
Cases[{{1, 2}, {3, a}, {b, 4, c}, {d, 2.5}},
      {(_Integer | _Symbol)...}]
{{1, 2}, {3, a}, {b, 4, c}}
```

Note, however, that the following syntax, while appealing, is not interpreted correctly:

```
Cases[{{1, 2}, {3, a}, {b, 4, c}, {d, 2.5}},
      {_(Integer | Symbol)...}]
{}
```

Here's why.

```
FullForm[_(Integer | Symbol)]
Times[Alternatives[Integer, Symbol], Blank[]]
```

6.4 Dynamic Programming

Certain problems have recursive solutions that are very natural but also are very inefficient. The reason for this inefficiency is that many identical recursive calls are made during any given computation. This property, called *overlapping subproblems*, is characteristic of many important problems. The most well-known examples are discrete optimization problems such as the 0-1 knapsack problem, optimal matrix-chain multiplication, finding the longest common subsequence of two sequences, and the Floyd-Warshall algorithm for finding shortest paths in a graph (see [Cormen et al. 90]). Other, non-optimization examples include convolution and multiple-class mean value analysis of queuing networks (see [Jain 91]).

Problems having the overlapping-subproblems property almost always are solved using *dynamic programming* [Cormen et al. 90], a catch-all term for any algorithm in which the definition of a function is extended as the computation proceeds. This is generally accomplished by constructing a solution "bottom up" (e.g., progressing from simpler to more complex cases), the goal being to solve each subproblem *before* it is needed by any other subproblem. The main disadvantage of dynamic programming is that it is often nontrivial to write code that evaluates the subproblems in the most efficient order.

However, there is an elegant dynamic programming technique that does not require the programmer to establish the evaluation order: recursion with *result caching* [sometimes called *memoization* (*sic*) in the computer science literature]. By caching the results of all recursive calls, the second and subsequent evaluations of any subproblem become constant-time operations, reducing the overall running time considerably. The ability to add rules to a function as the function executes makes result caching very easy to implement in *Mathematica*.

In this section we will solve two problems having the overlapping-subproblems property. The first of these, the computation of Fibonacci numbers, is a "toy" problem that we use merely to illustrate the concepts. The second problem we will consider is finding the optimal multiplication order for a chain of matrix multiplications, a problem having considerable practical significance. For an example of using this technique to implement multiple-class mean value analysis see [Wagner 95].

6.4.1 Fibonacci numbers

Consider the following function that generates the well-known *Fibonacci numbers:*

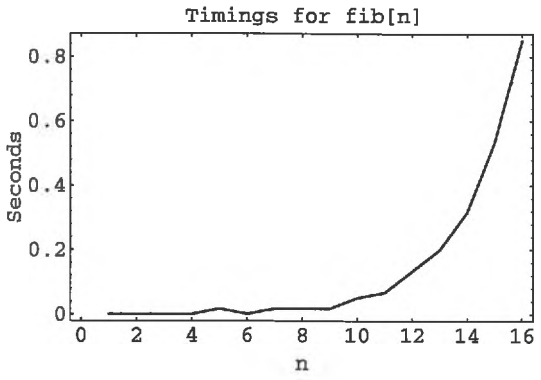
```
fib[n_] := fib[n - 1] + fib[n - 2]
fib[0] = fib[1] = 1;
```

Here are the first eight Fibonacci numbers.

```
Array[fib, 8]
{1, 2, 3, 5, 8, 13, 21, 34}
```

Unfortunately, the time required to calculate `fib[n]` is exponential in n:

```
t = Table[Timing[fib[n]][[1, 1]], {n, 1, 16}];

ListPlot[t, PlotJoined->True,
    PlotLabel->"Timings for fib[n]",
    Frame->True, FrameLabel->{"n", "Seconds"},
    FrameTicks->{Range[0, 16, 2], Automatic},
    DefaultFont->{"Times", 9}
];
```



A look at the execution trace of `fib` reveals the source of the inefficiency:

The second argument to `Trace` tells it to print only those intermediate expressions that match the pattern `fib[_]`.

```
Trace[fib[4], fib[_]]
{fib[4], {fib[3], {fib[2], {fib[1]}, {fib[0]}},
   {fib[1]}}, {fib[2], {fib[1]}, {fib[0]}}}
```

In this small example, `fib[3]` is evaluated once, `fib[2]` is evaluated twice (once during the call to `fib[4]` and once during the call to `fib[3]`), `fib[1]` is evaluated three times, and `fib[0]` is evaluated twice. In fact, the number of times `fib[1]` is called during the evaluation of `fib[n]` is equal to `fib[n-1]`:

This counts the number of times `fib[1]` occurs in the list returned by `Trace`.[6]

```
Table[Count[Flatten[Trace[fib[i], fib[_]]],
            HoldForm[fib[1]]],
      {i, 1, 12}
]
{1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144}
```

This is a classic example of the *overlapping-subproblems* property.

One way to solve this problem efficiently, which is quite straightforward in this simple case, is to perform the computation "bottom-up," i.e., use results for smaller arguments to calculate results for larger arguments in a monotonically increasing fashion:

The pure function takes `{fib[n-2], fib[n-1]}` as an argument and returns `{fib[n-1], fib[n]}`.

```
bufib[n_] :=
    Nest[{#[[2]], Plus @@ #}&, {0, 1}, n][[2]]
Table[bufib[n], {n, 0, 11}]
{1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144}
```

A more general and (in the author's opinion) elegant solution, which is the topic of the present section, is to cache the results of earlier computations.

In *Mathematica*, result caching is accomplished by a very modest change to the definition of a function.

```
Clear[fib]
fib[n_] := fib[n] = fib[n - 1] + fib[n - 2]
fib[0] = fib[1] = 1;
```

Comparing this definition of `fib` to the first one given, we see that the only difference is the prepending of `fib[n]=` to the right-hand side of the definition. Before we explain how this modification works, we give an example of its consequences.

Here is the rule set for `fib` before any evaluations are done.

```
?fib
Global`fib

fib[0] = 1
fib[1] = 1
fib[n_] := fib[n] = fib[n - 1] + fib[n - 2]
```

After evaluating `fib[3]` ...

```
fib[3]
3
```

there are more rules for `fib` than before!

```
?fib
Global`fib

fib[0] = 1
fib[1] = 1
fib[2] = 2
fib[3] = 3
fib[n_] := fib[n] = fib[n - 1] + fib[n - 2]
```

What is really going on here? When `fib[n_]` is matched with a particular value for the pattern variable `n` — say, *n0* — *Mathematica* evaluates the right-hand side of the definition. But the right-hand side is itself a call to `Set`, which results in the assignment of a value to the expression `fib[`*n0*`]`. From this time forward, whenever the value of `fib[`*n0*`]` is required, no recursive call is made.

---

6. It is necessary to wrap `HoldForm` around the pattern `fib[1]` because every element in the list returned by `Trace` is wrapped in `HoldForm`.

The new `fib` function is significantly faster.

```
Timing[fib[16]]
{0.0333333 Second, 1597}
```

Using the old definition of `fib`, this computation would, for all intents and purposes,[7] never finish.

```
Timing[fib[100]]
{0.133333 Second, 573147844013817084101}
```

Because the value of `fib[100]` has been cached, this call takes no time at all!

```
Timing[fib[100]]
{0. Second, 573147844013817084101}
```

Of course, result caching is a tradeoff of memory for time — there are now 102 rules defined for the symbol `fib`. If you evaluate a cached function for extremely large values of its arguments, you may run out of memory. And since the basis of the technique is recursion, it is not difficult to exceed `$RecursionLimit` on large input values.

A more subtle problem, which is likely to be encountered when solving optimization problems, is that the cached values created for one set of inputs are probably not correct for a different set of inputs (the next problem we consider is an example of this). Thus, the *Mathematica* programmer must provide an easy way for the user to reinitialize the cached functions. Finally, it must be pointed out that during the course of debugging such a function, care must be taken to always `Clear` the cached values and define the function from scratch whenever *any* change is made to the function.

Exercises

1. `Clear` the definition of `fib` (to remove any cached results) and reinitialize it. Now `Trace` the evaluation of `fib` for successively larger arguments (start small!) to see memoization in action. (Try using the expression `Trace[fib[n], _ = _]`.)
2. Try to evaluate `fib[10000]`. Can you program your way around this problem?

6.4.2 Application: Matrix-chain multiplication

The *matrix-chain multiplication* problem can be stated as follows: Given a chain (i.e., a sequence) of matrices whose dot product we wish to compute, parenthesize the chain so as to force the pairwise dot products to occur in an order that minimizes the number of scalar multiplications performed. Our presentation of this problem is modeled after [Cormen et al. 90] Section 16.1, and we quote results freely from that source. Readers interested in all of the details should consult [Cormen et al. 90].

The total number of scalar multiplications necessary to carry out a matrix-chain product can vary dramatically based on the parenthesization of the chain. Here's an example that shows how important the choice of parenthesization can be:

7. "For all intents and purposes" means about 10^8 years, give or take an order of magnitude.

Here are three matrices; we wish to compute `b1.b2.b3`.

```
b1 = Table[Random[], {300}, {10}];
b2 = Table[Random[], {10}, {300}];
b3 = Table[Random[], {300}, {10}];
```

There are two mathematically equivalent ways to compute `b1 . b2 . b3`: as `((b1 . b2) . b3)` or as `(b1 . (b2 . b3))`. In terms of computational effort, however, they are anything *but* equivalent. Note that the number of scalar multiplications required to compute the dot product of a $p \times q$ matrix with a $q \times r$ matrix is pqr. If the matrix product in this example were computed as `((b1 . b2) . b3)`, the total number of multiplications would be:

```
300 * 10 * 300 + 300 * 300 * 10
1800000
```

But if the product were computed as `(b1 . (b2 . b3))`, the number of scalar multiplications would be reduced by a factor of *30:*

```
10 * 300 * 10 + 300 * 10 * 10
60000
```

Now observe that *Mathematica*'s built-in function `Dot` is *not* smart enough to evaluate this product in the optimal order.

Here is how long it takes `Dot` to compute the example product.

```
b1 . b2 . b3; // Timing
{2.55 Second, Null}
```

Note the dramatic speedup if we force multiplication in the optimal order.[8]

```
Dot[b1, Dot[b2, b3]]; // Timing
{0.15 Second, Null}
```

This example motivates the need for an algorithm to determine the optimal matrix-chain multiplication order.

Suppose that the matrix chain to be multiplied is $A_1 \cdots A_n$, where A_i has dimensions $p_i \times p_{i+1}$. The problem we will solve is to (a) find the cost of multiplying out the matrix chain using an optimal parenthesization, and (b) produce a nested list of the indices $1, \ldots, n$ that indicates an optimal parenthesization, given the list of matrix dimensions `p = {p1, ..., pn+1}`. In the example given above, the list of dimensions is `{300, 10, 300, 10}`, the cost of an optimal evaluation is 60000, and an optimal parenthesization (the only one, in this case) is `{1, {2, 3}}`.

A brute-force approach to this problem, which consists of computing the cost of every possible parenthesization of the matrix chain, would take time that is exponential

8. It is necessary to use explicit calls to `Dot`, rather than parentheses, to effect the optimal multiplication order because the parser converts `b1 . (b2 . b3)` into `Dot[b1, b2, b3]`.

in the length of the chain (more precisely, it is at least as bad as $4^n/n^{3/2}$). This is computationally infeasible for all but the smallest values of n.

As a starting point in the search for a better solution, we note that the matrix-chain multiplication problem satisfies the *optimal substructure* property. Optimal substructure means that the optimal solution to the problem is built from the optimal solutions of smaller problems having the same structure as the original. For example, to find the optimal multiplication order for the matrix chain $A_1A_2A_3A_4A_5$, we must consider four alternative ways to split the original problem: $A_1(A_2A_3A_4A_5)$, $(A_1A_2)(A_3A_4A_5)$, $(A_1A_2A_3)(A_4A_5)$, and $(A_1A_2A_3A_4)A_5$. (There are only $n-1$ such splits in a chain of length n since matrix multiplication is noncommutative.) The optimal solution *given a particular split* must consist of optimal solutions to each of the two subproblems. Therefore, the subproblems have the same structure as the original one, but on a smaller scale.

This observation suggests a recursive formulation of the solution. If we denote by $m(i, j)$ the cost of optimally multiplying the matrix chain $A_i \cdots A_j$, then a recursive definition for $m(i, j)$ is:

$$m(i,j) = \begin{cases} 0 & \text{if } i = j \\ \min\limits_{i \le k < j} \left[m(i,k) + m(k+1,j) + p_i p_{k+1} p_{j+1} \right] & \text{if } i < j \end{cases} \quad (1)$$

This equation says that for any choice of where to "split" the chain (given by the index k), the total cost is equal to the cost of the optimal multiplication of all matrices to the left of the split ($m(i, k)$), plus the cost of the optimal multiplication of all matrices to the right ($m(k + 1, j)$), plus the cost of multiplying together these two intermediate results ($p_i p_{k+1} p_{j+1}$). The optimal cost, then, is the minimum cost over all of the $j - i$ possible choices for k.

It is clear that the matrix-chain multiplication problem suffers from overlapping subproblems as well. For example, each of the subproblems $A_1A_2A_3A_4$ and $A_2A_3A_4A_5$ requires the solution to the subproblem $A_2A_3A_4$. Likewise, $A_2A_3A_4A_5$ requires the solution to $A_3A_4A_5$, which is already being computed as part of $(A_1A_2)(A_3A_4A_5)$. The entire problem can be viewed as a pyramid-shaped directed graph in which the complete chain is at the apex and the individual pairings are at the base (Figure 6-1). The number of paths from the apex to any intermediate node in the graph is the number of times the solution to that subproblem will be required. [Cormen et al. 90] shows that the naïve recursive approach has a computational time complexity that is at least 2^n.

The dynamic programming approach to solving this problem is to compute the bottom row of the pyramid, $m(i, i + 1)$ for $i = 1, \ldots, n - 1$; then compute the second row from the bottom, $m(i, i + 2)$ for $i = 1, \ldots, n - 2$; and so on, until finally the apex $m(1, n)$ is computed. The computational time complexity of this approach is only proportional to n^3.

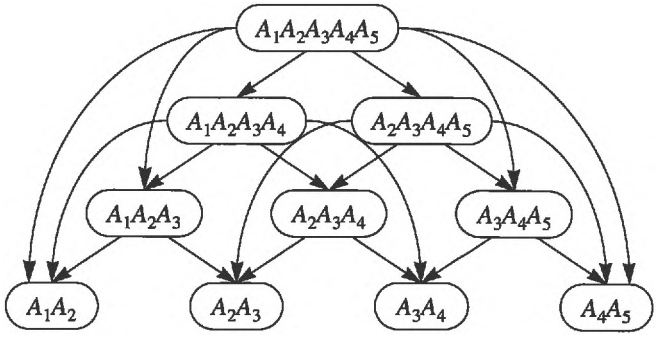


Figure 6-1 Computational structure of the matrix-chain multiplication problem.

In contrast to the bottom-up approach to dynamic programming, here is a "top-down" result-caching implementation of the recurrence equation for $m(i, j)$. Note that the form of the solution is a *direct* translation into a *Mathematica* expression of the recursive definition given in Equation (1). (The `m[__]` case is used whenever an expression such as `m[i, i]` is evaluated.)

```
m[i_, j_] /; i < j := m[i, j] =
Min[Table[m[i, k] + m[k + 1, j] +
            p[[i]] p[[k + 1]] p[[j + 1]],
          {k, i, j - 1}]
]
m[__] := 0
```

The result-caching implementation computes subproblems as shown in Figure 6-2 (the reader should attempt to verify this). For example, when the subproblem $A_2A_3A_4$ needs the solution to the subproblem A_3A_4, no work is done because the latter subproblem has been solved already by $A_3A_4A_5$.

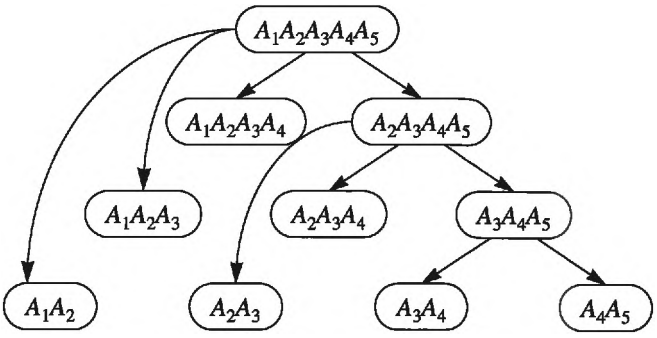


Figure 6-2 Result-cached computation of the matrix-chain multiplication problem.

Below, we solve an example problem from [Cormen et al. 90].

Here are the dimensions of the matrices in the chain.

```
p = {30, 35, 15, 5, 10, 20, 25};
```

A call to `m[1, 6]` returns the minimum number of scalar multiplications required for this chain.

```
m[1, 6]
15125
```

Here are all the intermediate results.

```
TableForm[Array[m, {5, 6}],
    TableHeadings->Automatic,
    TableAlignments->{Center, Right}
]
```

	1	2	3	4	5	6
1	0	15750	7875	9375	11875	15125
2	0	0	2625	4375	7125	10500
3	0	0	0	750	2500	5375
4	0	0	0	0	1000	3500
5	0	0	0	0	0	5000

The given algorithm just tells us the *cost* of the optimal multiplication order, but not what that order actually *is*. To construct the full solution, the `m` function needs to leave a "trail of bread crumbs" as it works. The modified version of `m` shown below stores the table of alternative costs in a local variable called `choices`. Then the index of the minimum cost (the optimal split) is stored in a global variable `s[i, j]`. Although purists may recoil at this use of side effects, it is defensible in this case for two reasons. First, the most obvious alternatives (having `m` return a list consisting of `{cost, position}`, or passing `s` as a *by-reference* parameter to `m`) make the code more complicated and less efficient. Second, if this code is encapsulated inside of a *package* (Chapter 8), `s` can be hidden inside of a private *context*, so these side effects will not be visible to the user.

Here we store the table of costs in a local variable called `choices`. Then the index of the minimum cost (the optimal split) is stored in `s[i, j]`.

```
Clear[m]
m[i_, j_] /; i < j := m[i, j] =
Module[{choices, best},
    choices =
        Table[m[i, k] + m[k + 1, j] +
                p[[i]] p[[k + 1]] p[[j + 1]],
              {k, i, j - 1}];
    best = Min[choices];
    s[i, j] = Position[choices, best][[1, 1]] + i - 1;
    best
]
m[__] := 0

m[1, 6]
15125
```

Here is the entire `s` table. (The rule for `s[__]` has been added for convenience.)

```
s[__] := 0
```

```
TableForm[
    Array[s, {5, 6}],
    TableHeadings->Automatic,
    TableAlignments->{Center, Right}
]
```

	1	2	3	4	5	6
1	0	1	1	3	3	3
2	0	0	2	3	3	3
3	0	0	0	3	3	3
4	0	0	0	0	4	5
5	0	0	0	0	0	5

The `s` table shows that the optimal split for the main problem is between the third and fourth matrix (`s[1, 6] = 3`). The optimal split for the subproblem $A_1A_2A_3$ is between matrices 1 and 2 (`s[1, 3] = 1`), and the optimal split for the subproblem $A_4A_5A_6$ is between matrices 5 and 6 (`s[4, 6] = 5`). We can use this information to generate a nested list of indices indicating the optimal parenthesization:

```
{1, 6} //. {i_Integer, j_Integer} /; i < j :>
            {{i, s[i, j]}, {s[i, j] + 1, j}}
{{{1, 1}, {{2, 2}, {3, 3}}},
  {{{4, 4}, {5, 5}}, {6, 6}}}

multorder = % /. {i_, i_} -> i
{{1, {2, 3}}, {{4, 5}, 6}}
```

Now that we have this list, how do we use it? First, suppose that we have a list of matrices of the given sizes:

```
A = Table[Random[], {i, 6}, {p[[i]]}, {p[[i + 1]]}];
```

We need to turn the parenthesized list of indices, `multorder`, into an expression of the following form:[9]

```
Dot[Dot[A[[1]], Dot[A[[2]], A[[3]]]],
    Dot[Dot[A[[4]], A[[5]]], A[[6]]]]
```

This transformation ought to be easy but it turns out to be slightly tricky. The obvious thing to do first is to change all of the heads in `multorder` from `List` into `Dot`. Unfortunately, that destroys the nested structure (which we worked so hard to concoct in the first place!) because `Dot` is `Flat`:

```
multorder /. List->Dot // FullForm
Dot[1, 2, 3, 4, 5, 6]
```

9. Note that we could not simply have had `m` compute the matrix product as it went along, because this would waste a lot of memory and time as nonoptimal multiplications (e.g., `A[[1]] . A[[2]]`) would get done along the way.

On the other hand, we can't substitute the matrices *before* changing `List` to `Dot` because the matrices themselves are nested lists!

We can get out of this "Catch-22" by changing the `List` heads in `multorder` to something else before substituting the matrices; afterward, we can go back and change the temporary heads to `Dot`. We'll use the symbol `naDot` (for "nonassociative `Dot`") as the temporary head.

```
multorder /. List->naDot
naDot[naDot[1, naDot[2, 3]], naDot[naDot[4, 5], 6]]
```

Next, we turn each index `i` into `A[[i]]` using a straightforward delayed-rule substitution:

The output is suppressed because this expression is huge.

```
chain = % /. i_Integer :> A[[i]];
Short[chain, 2]
naDot[naDot[{<<30>>}, naDot[{<<35>>}, {<<15>>}]],
  naDot[<<2>>]]
```

Now simply change `naDot` to `Dot` to evaluate the "parenthesized" chain. Since `Dot` is evaluated in the standard way, more deeply nested dot products will be evaluated before less deeply nested ones, thus preserving the "parenthesization."

Here is the time required for the optimally parenthesized matrix-chain product.

```
foo = chain /. naDot -> Dot; // Timing
{0.0666667 Second, Null}
```

For this example it is only slightly faster than passing the entire chain to `Dot`.

```
Timing[bar = Dot @@ A;]
{0.0833333 Second, Null}
```

Sanity check.

```
foo == bar
True
```

Techniques for evaluating this expression that do not require the use of a temporary head will be presented in Sections 7.3.6 and 7.3.8.

Exercises

1. Combine the results from this section to write a matrix-chain multiplication function that takes a sequence of matrices as an input and produces their product (in an optimal way, of course) as output. Note that you can find the sizes of the matrices by using `Dimensions`.
2. The *longest common subsequence* (LCS) problem is another problem that has a dynamic programming solution [Cormen et al. 90]. For purposes of this problem we will consider sequences to be lists of integers, although obviously the same approach generalizes to lists of any type of elements or even character strings.[10]

10. A character string can be converted to a list of integer codes by using the `ToCharacterCode` function.

The basic reasoning is: First, look at the last element in each sequence. If it is the same, then that element is part of the LCS and the problem reduces to finding an LCS for the pair of sequences minus their last elements. Otherwise, find the LCS of (i) the first sequence minus its last element and the second sequence, and (ii) the second sequence minus its last element and the first sequence. Whichever of these is longer is the LCS of the two original sequences.

Let $x=\{x_1, \ldots, x_m\}$ and $y=\{y_1, \ldots, y_n\}$ be sequences. Denote by `c[i, j]` the length of the LCS of the sequences $\{x_1, \ldots, x_i\}$ and $\{y_1, \ldots, y_j\}$. Then a recursive definition for `c[i, j]` is given by:

$$c[i,j] = \begin{cases} 0 & (i=0) \vee (j=0) \\ c[i-1,j-1]+1 & (i,j>0) \wedge (x[[i]] = y[[j]]) \\ \max(c[i,j-1], c[i-1,j]) & (i,j>0) \wedge (x[[i]] \neq y[[j]]) \end{cases}$$

Once you have computed the `c[i, j]` values, you then have to reconstruct the actual LCS. The procedure for doing so is, not surprisingly, recursive. Start by looking at x_m and y_n. If they are equal, then append that element to the LCS of $\{x_1, \ldots, x_{m-1}\}$ and $\{y_1, \ldots, y_{n-1}\}$. Otherwise, there are two cases to consider: If `c[m - 1, n] > c[m, n - 1]`, then return the LCS of $\{x_1, \ldots, x_{m-1}\}$ and y; else return the LCS of x and $\{y_1, \ldots, y_{n-1}\}$.

6.5 Overriding Built-in Functions

6.5.1 Protected symbols

Sometimes you want to override the behavior of one of the built-in functions. For example, suppose you have defined your own logarithm function, and you want it to have the property that `log[a b]` factors to `log[a] + log[b]`. If you try to modify the `Factor` function, you're in for a rude surprise.

```
Factor[log[x_ y_]] := log[x] + log[y]
SetDelayed::write:
   Tag Factor in Factor[log[(x_) (y_)]] is Protected.
$Failed
```

Most of the built-in symbols, such as `Factor`, have the attribute `Protected`, which prevents them from being modified inadvertently.

```
Attributes[Factor]
{Listable, Protected}
```

However, if you are resolute in your desire to add this rule to `Factor`'s repertoire, you can remove the `Protected` attribute and press onward.

```
Unprotect[Factor];
Factor[log[x_*y_]] := log[x] + log[y]
Protect[Factor];
```

A downvalue has been created for `Factor`.

```
DownValues[Factor]
{HoldPattern[Factor[log[(x_) (y_)]]] :>
   log[x] + log[y]}
```

More importantly, it actually works.

```
Factor[log[(x + 1)(2 x - 5)]]
log[1 + x] + log[-5 + 2 x]
```

Here's a situation in which our rule isn't applied:

```
Factor[1 + log[(x + 1)(2 x - 5)]]
1 + log[(1 + x) (-5 + 2 x)]
```

The reason is that the full form of this expression is `Factor[Plus[...]]`, which doesn't match our rule. This behavior is similar to that of the built-in `Factor` function, which doesn't "descend" into expressions looking for subexpressions to factor.

```
Factor[1 + Log[2 x^2 - 3 x - 5]]
1 + Log[-5 - 3 x + 2 x^2]
```

To factor lower-level expressions, you have to use either `MapAt` to target `Factor` to a particular subexpression, or `MapAll` to map `Factor` at every level of the expression.

```
MapAll[Factor, 1 + log[(x + 1)(2 x - 5)]]
1 + log[1 + x] + log[-5 + 2 x]
```

As a bonus, this technique also works in cases like the following:

```
MapAll[Factor, 1 + log[2 x^2 - 3 x - 5]]
1 + log[1 + x] + log[-5 + 2 x]
```

The reason this works is that `MapAll` operates "bottom up." Thus, `Factor` is applied to the polynomial before it is applied to the log of the polynomial.

Exercises

1. Create another rule for `Factor` so that an expression such as `log[(x + 1)^3]` factors to `3 log[x + 1]`.
2. Modify the `Dot` function so that when it is called with more than two arguments, it computes the optimal matrix-chain multiplication of the arguments (Section 6.4.2).

6.5.2 Upvalues

As an alternative to modifying a protected symbol, you can associate a rule for that symbol with certain other symbols that appear in the rule definition.

Before continuing, we have to remove any definitions we have added to the `Factor` symbol.

```
Unprotect[Factor];
Clear[Factor]
Protect[Factor];
```

This rule for factoring a log is not associated with `Factor`.

```
Factor[log[x_*y_]] ^:= log[x] + log[y]

DownValues[Factor]
{}
```

Rather, it is associated with the symbol `log`.

```
?log
Global`log

Factor[log[(x_)*(y_)]] ^:= log[x] + log[y]
```

The funny assignment operator `^:=` is called `UpSetDelayed`.[11] The example rule is not a downvalue for `log`, it is a new kind of rule called an *upvalue*:

```
{DownValues[log], UpValues[log]}
{{}, {HoldPattern[Factor[log[(x_) (y_)]]] :>
    log[x] + log[y]}}
```

The names are meant to reflect where in the definition a symbol is. If it's the head of the expression, it is "looking down" into the expression; if it's nested within the expression, it's "looking up" out of the expression.

The new definition works just like a `DownValue` for `Factor`.

```
Factor[log[(x + 2)(x - 5)]]
log[-5 + x] + log[2 + x]
```

Another way to define an upvalue is to use the `TagSetDelayed`[12] (`/: ... :=`) operator:

This is equivalent to the above use of `UpSetDelayed`.

```
log /: Factor[log[x_*y_]] := log[x] + log[y]
```

11. Naturally, there is a corresponding operator `UpSet` (`^=`) that evaluates the right-hand side of the definition before performing the assignment. Except for that difference, everything we have to say about one applies to the other. To keep things simple, we'll use `UpSet` as a generic term.
12. Likewise, there is a `TagSet` (`/: ... =`) operator; similar comments apply as for `UpSet`.

`TagSet` is more precise than `UpSet` in the following sense: `UpSet` creates an upvalue for every symbol that is at level `1` or is the head of an expression at level `1` in the rule definition.

Using `UpSet`, this rule creates `UpValues` for `g`, `h`, *and* `i`.

```
f[g[x_], h[y_, z_], i] ^:= ...
```

`TagSet`, on the other hand, allows you to specify the symbol for which the upvalue will be created:

Using `TagSet`, this rule creates an upvalue for `h` only.

```
h /: f[g[x_], h[y_, z_], i] := ...
```

Some people also prefer the "look" of a `TagSet` to the "look" of an `UpSet`; the former stands out better than the latter.

The motivation for using an upvalue is twofold. First, in the overwhelming majority of cases `Factor` will be called with arguments that do not involve the `log` function. In these cases the existence of the rule for `Factor[log[..]]` will not be checked, and so will not slow down the evaluation of `Factor`. (Imagine the overhead if a common function like `Factor` had to look at every rule for factoring every type of expression.)

Second, associating special rules for an individual function *with* that function helps to "localize the damage." It makes it easier to find all such rules when you need to work on the function some more at a future time. This argument is similar in spirit to the argument for programming in an object-oriented style. In fact, you can think of an `UpValue` as a kind of *virtual function* [Stroustrup 91]. This analogy seems especially appropriate since the kernel checks an expression for upvalues before checking it for downvalues, so that the former always override the latter.

Having said all that, we need to point out a limitation of upvalues: They can be associated only with symbols that are at level `1` or are the heads of expressions at level `1` in the rule definition. In other words, we can't make the following rule an upvalue for `log`:

```
log /: Factor[a_. + log[b_*c_]] :=
    a + log[b] + log[c]
TagSetDelayed::tagpos:
   Tag log in Factor[log[(b_) (c_)] + <<1>>]
     is too deep for an assigned rule to be found.
$Failed
```

The problem is that `log` is the head of an expression at level `2` in the rule definition. This limitation is necessary to keep expression evaluation relatively efficient: Without it, the kernel would have to check every symbol at all levels of an expression in order to decide how to evaluate the expression. In the given example, you have a choice of defining a downvalue for `Factor` or an upvalue for `Plus`. The former seems like the

logical choice, since `Factor` is much less common than `Plus`. Either way, you have to modify a protected symbol.

Finally, we need to mention one type of symbol that you can't modify, no matter how hard you try: Any symbol that has the attribute `Locked` in addition to the attribute `Protected` can't be modified. Only those symbols that are absolutely fundamental to the operation of *Mathematica* are `Locked`. An example of such a symbol is `List`.

```
Attributes[List]
{Locked, Protected}
```

Exercises

1. Define upvalues or downvalues to effect the following behaviors:

```
Factor[log[a^n]]
n log[a]

Expand[%]

log[a^n]

Expand[log[x] + log[y]]
log[x y]
```

2. Find all system symbols that are `Locked`. (*Hints:* The expression `Names["System`*"]` will return a list of the names (e.g., character strings) of all system symbols. Use `Select` with an appropriately defined predicate. The `Attributes` function works on names of symbols.)

6.5.3 Application: A default thickness for Plot

A question arose in the Internet newsgroup `comp.soft-sys.math.mathematica` concerning how one could change the default thickness of the lines used by the `Plot` command. One might suppose that the answer is simply to set the default option for `PlotStyle` using `SetOptions[PlotStyle, PlotStyle -> Thickness[n]]`. Unfortunately, this will not work if the user specifies an explicit `PlotStyle` option in a call to `Plot`, even if that `PlotStyle` does not specify a `Thickness` (e.g., `Plot[f[x], {x, a, b}, PlotStyle -> RGBColor[1, 0, 0]]`). No specific option for plot thickness exists, and no global variable exists for this purpose either (e.g., as `$DefaultFont` does for plot fonts).

It's possible to solve this problem by adding a few new rules to the `Plot` command, but getting it right is tricky. One must realize that:

(a) `PlotStyle` may be a single graphics directive, or it may be a list.

(b) If `PlotStyle` is a list, its elements may themselves be either single graphics directives or lists of such.

(c) `PlotStyle` may be omitted entirely.

(d) The default should defer to `Thickness` directives specified by the user.

(e) The default should continue to work even if the default `PlotStyle` option is changed with `SetOptions`.

First and foremost, we should plan ahead: The default thickness should be easy to change. This can be accomplished by defining a new global variable, `$DefaultThickness`, that we use in our new rules.

```
$DefaultThickness = .004;
```

Second, we must `Unprotect` the `Plot` command:

```
Unprotect[Plot];
```

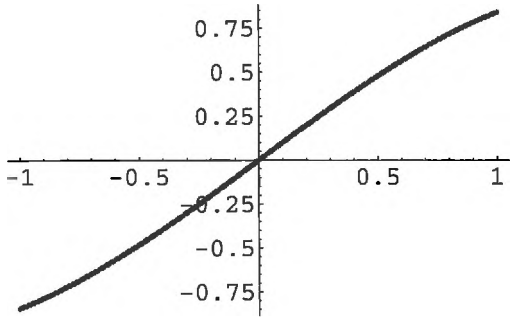
Now we are ready to enter the new rules for `Plot`. Dealing with the case in which `PlotStyle` is a single graphics directive is easy:

```
Plot[heads___, PlotStyle->style_, tails___] /;
  !ListQ[style] && FreeQ[style, Thickness] :=
    Plot[heads,
         PlotStyle->{{style,
                      Thickness[$DefaultThickness]}},
         tails]
```

The use of `BlankNullSequence` patterns (triple underscore) at the beginning and end of the parameter list makes this rule work no matter where the `PlotStyle` option appears. `!ListQ[style]` checks that `style` is a single graphics directive. `FreeQ[style, Thickness]` ensures that we do not override a manifest `Thickness` directive (point d above). It will appear, in some form, in nearly all of the rules that we define. This rule replaces `style` with `{{style, Thickness[$DefaultThickness]}}`. Here is an example:

`PlotStyle->gray` is changed to `PlotStyle->{{gray, Thickness[.01]}}`.

```
gray = GrayLevel[.5]; dashed = Dashing[{.04, .03}];
$DefaultThickness = .01;
Plot[Sin[x], {x, -1, 1}, PlotStyle->gray];
```



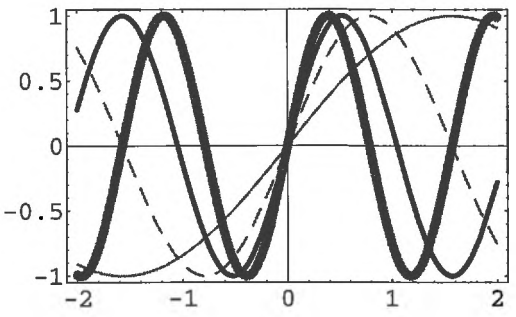
Dealing with the case in which `PlotStyle` is a list is only slightly more difficult:

```
Plot[heads___, PlotStyle->{h___, style_, t___},
     tails___] /; FreeQ[style, Thickness] :=
    Plot[heads,
         PlotStyle->
             {h, Flatten[{style,
                     Thickness[$DefaultThickness]}],
              t},
         tails]
```

Destructuring does most of the work. First of all, this rule cannot match unless `PlotStyle` is a list. When this is the case, each element of the list is a style or list of styles for one particular curve. This rule matches each individual curve style in turn with the pattern variable `style`; if `style` does not contain a `Thickness` directive (`FreeQ[style, Thickness]`), one is added. `Flatten` is used so that it is irrelevant whether `style` is a single graphics directive or a list of them.

In this example some of the plot styles are single directives and others are lists of directives. In either case, those styles for which no `Thickness` is specified are given a thickness value of `$DefaultThickness`; the others are left as they are.

```
$DefaultThickness = .005;
Plot[{Sin[x], Sin[2 x], Sin[3 x], Sin[4 x]}, {x, -2, 2},
    PlotStyle->{gray,
                {gray, dashed},
                Thickness[.01],
                {gray, Thickness[.02]}},
    Frame->True
];
```



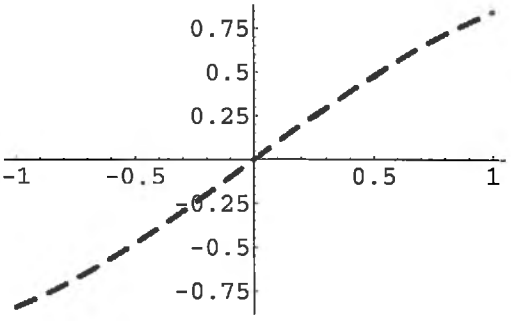
Now we must take care of the case in which the user changes the default `PlotStyle` option. The easiest way to do this is to add a rule that inserts the default `PlotStyle` option when no such option is present. (The default value of the option can be found using the expression `PlotStyle /. Options[Plot]`.) Subsequently, one of the rules defined earlier will fire if the inserted `PlotStyle` has no `Thickness` primitive.

`Evaluate` is necessary because `Plot` has the attribute `HoldAll`.

```
Plot[args__] /; FreeQ[{args}, PlotStyle] :=
    Plot[args,
      Evaluate[PlotStyle->(PlotStyle /. Options[Plot])]
    ]
```

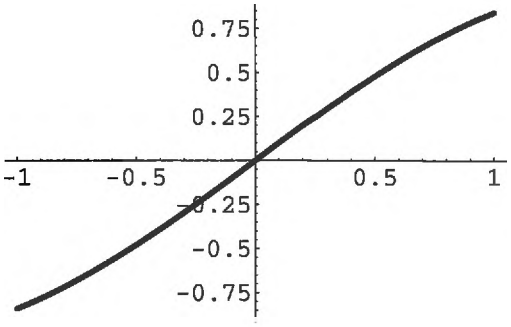
This seems to work.

```
SetOptions[Plot, PlotStyle->{{gray, dashed}}];
$DefaultThickness = .01;
Plot[Sin[x], {x, -1, 1}];
```



But look at what happens if the default `PlotStyle` is `Automatic`.

```
SetOptions[Plot, PlotStyle->Automatic];
Plot[Sin[x], {x, -1, 1}];
```




```
Graphics::gprim:
   Unknown Graphics primitive Automatic encountered.
```

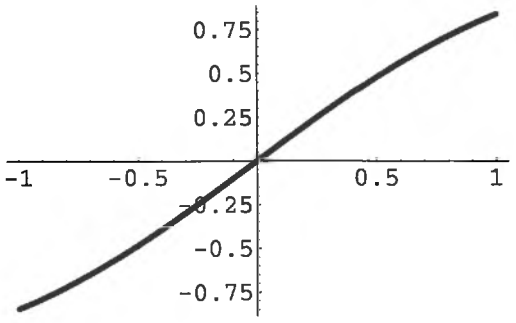
What went wrong? Our last rule inserted the `PlotStyle->Automatic` option in this plot command, and since the right-hand-side of this rule does not contain `Thickness`, another one of our rules appended the `Thickness` directive to `Automatic`. The net result was `PlotStyle->{Automatic, Thickness[.01]}`, which gave rise to the error message. What we need is for the `Thickness` directive to *replace* the word `Automatic`. It should be easy to fix:

This rule traps the problematic case.

```
Plot[heads___, PlotStyle->Automatic, tails___] :=
    Plot[heads,
      PlotStyle->Thickness[$DefaultThickness], tails]
```

Now what?!

```
Plot[Sin[x], {x, -1, 1}];
```




```
Graphics::gprim:
   Unknown Graphics primitive Automatic encountered.
```

In a case such as this, one should suspect that the rules for the function are not in the correct order. This is easily checked:

We show only the left-hand side of each downvalue for the sake of brevity.

```
First /@ DownValues[Plot]
{HoldPattern[Plot[heads___, PlotStyle -> style_,
     tails___] /;
    !ListQ[style] && FreeQ[style, Thickness]],
  HoldPattern[Plot[heads___,
     PlotStyle -> {h___, style_, t___}, tails___] /;
    FreeQ[{style}, Thickness]],
  HoldPattern[Plot[args__] /; FreeQ[{args}, PlotStyle]],
  HoldPattern[Plot[heads___, PlotStyle -> Automatic,
    tails___]]}
```

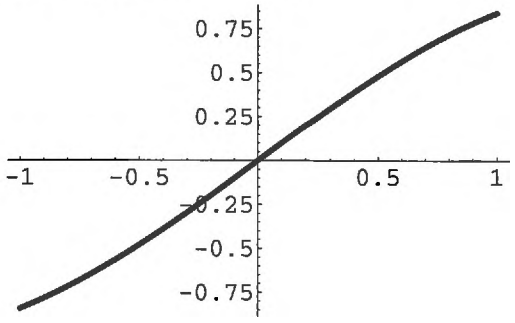
Confirmed: The rule for `PlotStyle` matching a generic single item gets checked before the rule for matching the keyword `Automatic`. We need to reorder the rules manually.

We can reorder the rules correctly by rotating `DownValues[Plot]` to the right one place, which will make the last rule we added "rise to the top."

```
DownValues[Plot] = RotateRight[DownValues[Plot]];
First /@ DownValues[Plot]
{HoldPattern[Plot[heads___, PlotStyle -> Automatic,
    tails___]], HoldPattern[Plot[heads___,
     PlotStyle -> style_, tails___] /;
    !ListQ[style] && FreeQ[style, Thickness]],
  HoldPattern[Plot[heads___,
     PlotStyle -> {h___, style_, t___}, tails___] /;
    FreeQ[{style}, Thickness]],
  HoldPattern[Plot[args__] /; FreeQ[{args}, PlotStyle]]}
```

That eliminated the error message.

```
Plot[Sin[x], {x, -1, 1}];
```



Finally, reprotect the `Plot` command.

```
Protect[Plot]
{Plot}
```

Exercise

1. Add a default `PointSize` to the `ListPlot` function.

6.6 Additional Resources

Almost every book about *Mathematica* is replete with programming examples, and some books are dedicated entirely to the subject. Two that stand out for their technical depth are by Maeder ([Maeder 91], [Maeder 94a]). Neither one is recommended for beginners, although readers of this book ought to be able to tackle them after finishing this chapter.

Another book devoted entirely to *Mathematica* programming, but at the level of an introductory text, is [Gaylord et al. 93].

[Gray 94] contains several nontrivial examples of using rule-based programming to implement knowledge bases.

7

Expression Evaluation

The most important function of the *Mathematica* kernel is to evaluate expressions to produce new expressions. We already know what expressions are, but what, exactly, does it mean to *evaluate* an expression? Until now we have been content with the somewhat vague and definitely oversimplified description, "The kernel applies rules to the expression until the expression stops changing." However, the evaluation process is much deeper than this. Aside from the fact that there are six different kinds of rules, there also are a number of *attributes* that affect the evaluation process, as well as an assortment of symbols that receive special treatment. A thorough comprehension of expression evaluation will greatly increase your abilities as a *Mathematica* programmer, and furthermore will enable you to understand the occasional unexpected result that most *Mathematica* users find inscrutable.

Section 7.1, "The Evaluation Process," is a description of the evaluation process in all its generality. Toward the end of that section some special cases that are not widely understood will be examined in greater detail. Section 7.2, "Nonstandard Evaluation," focuses on the aspects of the evaluation process that are exceptions to the general rule of evaluating all parts of an expression completely. Section 7.3, "Working with Held Expressions," puts the knowledge gained in Section 7.2 to practical use.

7.1 The Evaluation Process

7.1.1 Rules are associated with symbols

Every value or function definition entered into a *Mathematica* session is stored as a *global rule*, that is, a rule that is matched against every expression encountered by the kernel. For example, a definition like this one:

```
a := b + c
```

is stored as a type of rule called an *ownvalue:*

```
OwnValues[a]
{HoldPattern[a] :> b + c}
```

This rule simply says that anytime the kernel encounters the symbol `a`, it can be rewritten as the expression `b + c`. (The `HoldPattern` symbol in the left-hand side of the rule will be explained in Section 7.2.5. Users of version 2.2 and earlier will see the symbol `Literal` instead of `HoldPattern`.)

To take another example, a "function definition" such as the one shown below is also stored as a rule; in this case, the rule is called a *downvalue:*

```
f[x_] := x^2
DownValues[f]

{HoldPattern[f[x_]] :> x^2}
```

This rule says that anytime the kernel encounters an expression that matches the pattern `f[x_]` (`x_` is a pattern that represents any expression), the entire expression can be replaced by the new expression `x^2` (where `x` denotes not the literal symbol `x`, but rather a placeholder for the matched subexpression).

Every global rule is associated with some symbol, which is called the rule's *tag*. In the first example above, the tag is the symbol `a`, and in the second example it is the symbol `f`. All told, there are six different kinds of rules that can be associated with a tag; they are listed in Table 7-1. Any of the six kinds of rules can be stored as *Mathematica* expressions, or they can be defined in code internal to the kernel. Note that some system-defined symbols are implemented partially or entirely by external rules

Table 7-1 The Six Types of Rules for a Symbol

Type of Rule	For Evaluation of:	Example Definition
`OwnValues[sym]`	`sym`	`sym :=`
`DownValues[sym]`	`sym[...]`	`sym[...] :=`
`UpValues[sym]`	*head*`[..., sym, ...]` or *head*`[..., _sym, ...]`	*head*`[..., sym, ...] ^:=` or `sym /:` *head*`[..., sym, ...] :=`
`SubValues[sym]`	`sym[...][...]`	`sym[...][...] := ...`
`NValues[sym]`	`N[sym, precision]` or `N[sym[...], precision]`	`N[sym[...],` *precision*`] := ...`
`FormatValues[sym]`	*format*`[sym[...]]`[a]	`Format[sym[...],` *format*`] :=`

a. *format* can be any print-formatting function such as `TeXForm`, `CForm`, etc.; it defaults to `OutputForm`.

(although it usually is necessary to remove the symbol's `ReadProtected` attribute in order to see these rules). The external rules always take precedence over the internal ones, as we shall see shortly.

An upvalue is used to associate a rule with a symbol that is not the head of the expression to which the rule is matched. The most common use for upvalues is to add special cases to built-in functions. For example, let `omega` denote an exact solution to the transcendental equation `omega == Exp[omega]`. Then the following definition, which is an example of an `UpSet` operation (see Section 6.5.2, "Upvalues"), could be created to represent this fact:

```
Exp[omega] ^:= omega

Exp[omega]
omega
```

The `Exp` function has not been modified; instead, this rule is stored as an upvalue for the symbol `omega`:

```
UpValues[omega]
{HoldPattern[Exp[omega]] :> omega}
```

In cases in which an expression matches both an upvalue and a downvalue, the upvalue always takes precedence.

Nvalues can be used to specify what should happen to an expression when the `N` (numerical evaluation) operator is applied to it. An appropriate use for an Nvalue in the present context would be to define a numerical conversion rule for the `omega` symbol:

```
N[omega, p_:$MachinePrecision] :=
    FindRoot[x == Exp[x], {x, 1 + I},
        AccuracyGoal->p,
        WorkingPrecision->p + 10][[1, 2]]
```

Although this looks rather like a downvalue for `N`, the tag for this rule is, in fact, `omega`:

```
NValues[omega]
{HoldPattern[N[omega, p_:16]] :>
   FindRoot[x == Exp[x], {x, 1 + I},
     AccuracyGoal -> p,
     WorkingPrecision -> p + 10][[1, 2]]}
```

We have now succeeded in defining an exact representation of a transcendental number that behaves much as built-in transcendental constants such as `E` and `Pi` do, in that it simplifies symbolically when possible and evaluates numerically only when forced to:

```
omega - Exp[omega]
0
```

```
N[omega, 20] - Exp[N[omega, 20]]
```

$$0.\ 10^{-21} + 0.\ 10^{-20}\ I$$

The interplay between the numerical evaluation operator `N` and other definitions is quite involved; a full discussion is deferred to Section 9.3, "Numerical Evaluation."

Format values are used to change the way the kernel prints certain expressions. For example, here is a traditional-looking output format for the `BesselJ` function.

```
Unprotect[BesselJ];
Format[BesselJ[n_, x_]] := Subscripted[J[n]][x]
Protect[BesselJ];

BesselJ[1, Pi/2]
```

$$J_1\left[\frac{Pi}{2}\right]$$

Format values will be discussed in greater detail in Section 9.4, "Custom Output Formats."

Subvalues are rather obscure and will not be discussed beyond the coverage in this section and the next. Subvalues are so uncommon that they are not even documented in *The Mathematica Book*. The purpose of subvalues seems to be to define functions that return other functions. In fact, the built-in function called `Function`, which is used to define pure functions, is a perfect example of this.

```
Function[x, x^2][y + z]
```

$$(y + z)^2$$

By itself, the expression `Function[x, x^2]` is a completely evaluated expression. It is only when this expression is used as the head of another expression that it causes any term rewriting to take place.

Some other built-in functions that have predefined subvalues are `InterpolatingFunction`, `InverseFunction`, and `Derivative`, although, like `Function`, their subvalues are implemented internally and cannot be inspected.

External rules can be inspected and changed by the programmer using normal *Mathematica* list manipulation mechanisms. For example,

Create two downvalues for `f`.

```
Clear[f]
f[x_?EvenQ] := "even"
f[x_?OddQ] := "odd"
Array[f, 6]
{odd, even, odd, even, odd, even}
```

```
DownValues[f]
{HoldPattern[f[(x_)?EvenQ]] :> even,
  HoldPattern[f[(x_)?OddQ]] :> odd}
```

You can modify `DownValues[f]` directly.

```
DownValues[f] = Drop[DownValues[f], -1]
{HoldPattern[f[(x_)?EvenQ]] :> even}
```

The rule for `f[x_?OddQ]` is gone.

```
Array[f, 6]
{f[1], even, f[3], even, f[5], even}
```

The order in which the rules are stored within, e.g., `DownValues[f]` determines the order in which the kernel tries to apply them. Usually the kernel is smart enough to store rules in the intended order; often, however, when functions have lots of rules, or several rules of equal complexity, it's just not possible for the kernel to guess the programmer's intentions. (A nontrivial example of this was seen in Section 6.5.3.)

For example, the function defined below has two rules with identical structures.

```
Clear[f]
f[n_] := n
f[n_] := n/3 /; Mod[n, 3] == 0
f[n_] := n/2 /; Mod[n, 2] == 0
```

As a result, the kernel stores those rules in the order in which they are entered. On the other hand, the first rule, which the kernel recognizes as being more general than the other two, is moved to the end of `DownValues[f]` so that it is tried last.

Note that the kernel has reordered the rules.

```
DownValues[f]
{HoldPattern[f[n_]] :> n/3 /; Mod[n, 3] == 0,
  HoldPattern[f[n_]] :> n/2 /; Mod[n, 2] == 0,
  HoldPattern[f[n_]] :> n}
```

`f[6]` evaluates to 2 rather than 3 because of the order of the rules.

```
Array[f, 10]
{1, 1, 1, 2, 5, 2, 7, 4, 3, 5}
```

You can change the order in which rules are applied by rearranging `DownValues[f]` manually.

This expression permutes `DownValues[f]`.

```
DownValues[f] = DownValues[f][[{2, 1, 3}]]
{HoldPattern[f[n_]] :> n/2 /; Mod[n, 2] == 0,
  HoldPattern[f[n_]] :> n/3 /; Mod[n, 3] == 0,
  HoldPattern[f[n_]] :> n}
```

Now `f[6]` evaluates to 3 rather than 2.

```
Array[f, 10]
{1, 1, 1, 2, 5, 3, 7, 4, 3, 5}
```

This technique is especially useful when you need to override the behavior of a built-in function that has externally defined rules. By suitably reordering the `DownValues` for the function, you can ensure that your rules are tried first.

7.1.2 Properties that affect the evaluation process

The evaluation process is affected in certain ways by properties that are associated with symbols. The properties that can affect the evaluation of an expression with head `sym` are `Attributes[sym]` and `DefaultValues[sym]`. Default values (Section 6.3.2) are used in pattern matching, and will not be discussed here. The different kinds of attributes are listed in Table 7-2. Many of these will be discussed throughout this chapter.

Table 7-2 Attributes

Attribute	Meaning	Example Symbols
`Constant`	All derivatives are 0	`E`, `GoldenRatio`, `Pi`
`Flat`	Associativity	`And`, `Dot`, `Join`, `Max`, `Plus`
`HoldAll`	All arguments are unevaluated	`And`, `Clear`, `If`, `Plot`, `While`, `Protect`, `SetDelayed`
 `HoldAllComplete`	No modification of any kind to any argument	`BoxForm`, `HoldComplete`, `Unevaluated`
`HoldFirst`	First argument is unevaluated	`AppendTo`, `Increment`, `Set`
`HoldRest`	All but first argument are unevaluated	`RuleDelayed`, `Save`
`Listable`	Automatically threaded over lists	All numerical functions
`Locked`	Attributes cannot be changed	`$Aborted`, `I`, `List`, `Symbol`, `True`, `False`
 `NHoldAll`[a]	Arguments are unaffected by `N`	`Root`
`NHoldFirst`[a]	First argument is unaffected by `N`	`EllipticTheta`, `PolyGamma`
`NHoldRest`[a]	All but first argument are unaffected by `N`	None
 `NumericFunction`	`f[__?NumericQ]` is numeric	All numerical functions
`OneIdentity`	`f[a]`, `f[f[a]]`, etc. are equivalent to `a` for purposes of pattern matching	`And`, `Dot`, `Join`, `Max`, `Plus`
`Orderless`	Commutativity	`And`, `Max`, `Plus`

Table 7-2 (*Continued*) Attributes

Attribute	Meaning	Example Symbols
`Protected`	Values cannot be changed	Most system symbols
`ReadProtected`	Values cannot be read	`CosIntegral`, `FullOptions`, `PowerExpand`
`SequenceHold`	Arguments with head `Sequence` are not flattened	`Rule`, `Set`, `Timing`
`Stub`	`Needs` is called automatically when the symbol is used	Symbols defined in master packages
`Temporary`	Local variable	Symbols defined in modules

a. Although `NHoldAll`, `NHoldFirst`, and `NHoldRest` are new to version 3.0, identical functionality was available in version 2.2 through the *undocumented* attributes `NProtectedAll`, `NProtectedFirst`, and `NProtectedRest` (although none of the built-in functions used them). Programmers who like to live on the edge, take note: These undocumented functions have been dropped in version 3.0.

There are two other kinds of properties that can be associated with symbols, but they do not affect the evaluation process. These are `Options[sym]`, which returns a list of all options for a symbol, and `Messages[sym]`, which returns a list of all messages of the form *`sym::msgname`*. Although it may seem as though options affect the evaluation process, we need to draw a distinction between what is done by the kernel and what is done by the code defined for a symbol. Options are *ignored* by the kernel; it is up to the programmer of a function to see to it that the options are used by that function. We illustrated the use of options in Section 6.3.5; we shall go into greater detail about the proper way to use options and messages in Chapter 9, "Details, Details."

Exercise

1. Find all system-defined functions that are `Flat` but not `Orderless`. (*Hint:* Use `Names["System`*"]` to get a list of names of all system symbols, and use `Select` on the resulting list.)

7.1.3 The main evaluation loop

The main evaluation loop executed by the *Mathematica* kernel is summarized below. Most of this information is from [Withoff 93], with some explanatory comments and version 3.0-specific features added by the author. Although this summary is rather dry, the entire remainder of this chapter is devoted to explaining the nuances of the algorithm.

The first few steps of the evaluation process deal with atomic expressions:

1. If the expression is a string or a number, return it. (Strings and numbers evaluate to themselves.)
2. If the expression is a symbol that has no ownvalue, return it.
3. If the expression is a symbol that has an ownvalue, replace it with the ownvalue and start over with the new expression. (In general, if the expression being evaluated changes at any step, then the process begins again with the new expression.)

At this point the expression being evaluated must be a normal expression, i.e., an expression of the form h`[`$part_1$`, `$part_2$`, ...]`:

4. If no part of the expression has changed since the last evaluation of it, return the expression. This is an optimization that prevents unnecessary reevaluation of large expressions.[1]
5. Call the evaluation routine recursively on h. This is necessary because h itself could evaluate to something else. For example, h might be a symbol that evaluates to a pure function; h could even be another normal expression (e.g., in the expression `Derivative[2][f][x]`, the head is `Derivative[2][f]`, which may evaluate to a pure function for certain values of `f`).

At this point the head h has been determined. (Note that h may be a symbol or a normal expression.) Now various actions that depend on the attributes of h are taken:

6. If h has the `HoldAllComplete` attribute, skip to step 14.

7. For each $part_i$,
   (a) If $part_i$ is of the form `Evaluate[`*expr*`]`, remove the `Evaluate` head and call the evaluation routine recursively on *expr*. Continue with step 7.
   (b) If $part_i$ is of the form `Unevaluated[`*expr*`]`, replace $part_i$ by *expr* and make a note that this was done (see step 16). Continue with step 7.
   (c) If h is a symbol having the `HoldAll` attribute, continue with step 7.
   (d) If i `==` 1 and h is a symbol having the `HoldFirst` attribute, continue with step 7.
   (e) If i `>` 1 and h is a symbol having the `HoldRest` attribute, continue with step 7.
   (f) Call the evaluation routine recursively on $part_i$.

In other words, parts having the head `Evaluate` are always evaluated, and parts having the head `Unevaluated` are never evaluated. Evaluation of other parts is controlled by the `Hold-` attributes, with the default being that parts are evaluated.

---

1. There exist pathological cases in which this optimization may prevent an expression from reevaluating when, in fact, that expression depends on symbols that have changed. You can force such changes to propagate using the `Update` function; see §2.5.12 of *The Mathematica Book*.

8. If *h* is `Flat`, then splice the elements of any part with head *h* into the sequence of parts, e.g., `f[a, f[b, c], d]` ⇒ `f[a, b, c, d]`. Associative operators such as `Plus`, `Times`, and `Dot` have the `Flat` attribute.

9. If *h* does *not* have the attribute `SequenceHold`, then splice the elements of any part with head `Sequence` into the sequence of parts, e.g., `f[a, Sequence[b, c], d]` ⇒ `f[a, b, c, d]`. All of the assignment operators have the `SequenceHold` attribute (version 3.0 only).
10. If *h* is `Listable`, then thread *h* over any of the $part_i$ that are lists, e.g., `f[{a, b}, c, {d, e}]` ⇒ `{f[a, c, d], f[b, c, e]}`. All numerical functions, predicates, and selected symbolic functions have the `Listable` attribute (see Table 3-1 on page 67).
11. If *h* is `Orderless`, then sort the $part_i$, e.g., `f[c, b, a]` ⇒ `f[a, b, c]`. Commutative operators such as `Plus` and `Times` have the `Orderless` attribute.

The head and the parts of the expression all have been evaluated by now (to the extent allowed by the attributes of *h*). Actual rule application begins here:

12. Apply external upvalues attached to the *symbolic heads* of the $part_i$. The symbolic head of an expression is the expression itself, if the expression is a symbol; the head of the expression, if the head is a symbol; the head of the head, etc. For example, the symbolic head of `Derivative[2][f][x]` is `Derivative`.
13. Same as step 12, using internally defined upvalues.
14. If *h* is a symbol, apply user-defined downvalues. If *h* is not a symbol, apply user-defined subvalues.
15. Same as step 14, using internally defined downvalues or subvalues.

Finally, there's a bit of tidying up to do:

16. If no applicable rules were found and any of the $part_i$ had the head `Unevaluated`, restore that head.
17. Discard the head `Return`, if present.

7.1.4 Observations

A solid understanding of the evaluation process is one of the crucial characteristics that separates casual *Mathematica* programmers from *power programmers*. This understanding, combined with judicious use of `Trace` (discussed in detail in Chapter 13, "Debugging), `FullForm`, `Hold`, and related functions, makes it possible to figure out about 99.99 percent of the mystifying results that the kernel occasionally produces.[2]

In this section we'll give some examples of less obvious consequences of the algorithm outlined in the previous section. Along the way, it is hoped that you will have an

2. For the remaining 0.01 percent, contact WRI technical support. You may have found a bug!

occasional "light bulb" experience, as you suddenly come to understand something that previously was a mystery to you.

One thing we won't discuss much in this section is nonstandard evaluation; this topic is rich enough and important enough to merit two main sections of its own, coming up after this one.

Evaluation of heads

The first observation we shall make is that the head of an expression evaluates before any of its parts. Moreover, the head evaluates completely — in other words, the evaluation loop is called recursively on the head before continuing. Consider the following example:

Here are some test functions.

```
Clear[f, g, h]
f[x_] := 2 x
g[x_] := 3 x
h[x_] := 4 x
```

Now set up ownvalues for f and g.

```
f = g;
g = h;
```

f evaluates to g, which then evaluates to h, before the argument is evaluated.

```
Clear[a]
Trace[f[2a - a]]
{{f, g, h}, {2 a - a, -a + 2 a, a}, h[a], 4 a}
```

The braces in the trace are telling: `{f, g, h}` is one complete subevaluation. In contrast, if the evaluation loop were not recursive, then after f was replaced by g it could be the case that the rule for `g[x_]` would fire — which clearly does not happen. The consequence of the recursive evaluation of heads is that the only thing that matters for the rest of the evaluation process is the *final* value of the head. In particular, if f or g had attributes, those attributes would have had no effect on the computation.

Here, f is given the `Orderless` attribute.

```
Attributes[f] = {Orderless};
```

The arguments are not reordered because the attributes of h are the only ones that matter.

```
f[5, 4, 3, 2, 1]
h[5, 4, 3, 2, 1]
```

On the other hand, if the ownvalue for f is removed, `Orderless` "does its thing."

```
f =.
f[5, 4, 3, 2, 1]
f[1, 2, 3, 4, 5]
```

Evaluation of pure functions

While we are on the topic of heads and attributes, we should mention a related phenomenon that has to do with pure functions. When a pure function is assigned to a sym-

bol, the behavior of that symbol is not quite the same as if an equivalent downvalue were attached to the symbol. First of all, no pattern matching is done on a pure function.

Define two similar functions: one as a rule, the other as a pure function.

```
ClearAll[f]
f[x_] := x^2
pf = Function[x, x^2];
```

On the surface, they appear to behave identically.

```
{f[a], pf[a]}
```

$$\{a^2, a^2\}$$

But `pf` will evaluate no matter what gets passed to it.

```
{f[a, b], pf[a, b]}
```

$$\{f[a, b], a^2\}$$

Note that there is no rule for `f` that matches the expression `f[a, b]`, so that expression does not evaluate. On the other hand, `pf[a, b]` evaluates with # interpreted as #1, thus squaring the first of the two arguments.

Pattern matching is not performed on pure functions because the evaluation of a pure function is fundamentally different than the evaluation of a "normal" function. Examine the following trace:

```
Trace[pf[a]]
```

$$\{\{pf, Function[x, x^2]\}, Function[x, x^2][a], a^2\}$$

The symbol `pf` is replaced by its ownvalue, yielding the intermediate form `Function[x, x^2][a]`. Thus, the symbol `pf` is completely out of the picture before the function evaluation takes place. The transformation from the intermediate form to the final answer, `a^2`, is effected by built-in subvalues for the symbol `Function`, not by downvalues for `pf` (in fact, `pf` has no downvalues).

The previous example may make the following, more subtle difference between ordinary and pure functions easier to understand.

`f` and `pf` now take two arguments; both symbols are given the attribute `Orderless`.

```
Clear[f, pf]
Attributes[f] = Attributes[pf] = {Orderless};
f[x_, y_] := {x, y}
pf = Function[{x, y}, {x, y}];
```

The attribute has no effect on `pf`!

```
{f[2, 1], pf[2, 1]}
{{1, 2}, {2, 1}}
```

Here's why: `pf` evaluates to the pure function (which has no attributes) before attributes are checked.

```
Trace[pf[2, 1]]
{{pf, Function[{x, y}, {x, y}]},
  Function[{x, y}, {x, y}][2, 1], {2, 1}}
```

Thus we must distinguish between the attributes of a *symbol* — which matter only when that symbol is the fully evaluated form of an expression's head — and the

attributes of a pure function. A mechanism does exist for giving attributes to a pure function: An attribute (or a list of attributes) can be specified as an optional parameter to `Function`. (Note that there's no special input form for this feature, i.e., it cannot be used with the #-& style of pure functions.)

This syntax gives the pure function the `Orderless` attribute.

```
ClearAll[pf]
pf = Function[{x, y}, {x, y}, Orderless];
```

After the evaluation of the symbol `pf`, the arguments to the pure function are sorted.

```
Trace[pf[2, 1]]
{{pf, Function[{x, y}, {x, y}, Orderless]},
  Function[{x, y}, {x, y}, Orderless][2, 1],
  Function[{x, y}, {x, y}, Orderless][1, 2], {1, 2}}
```

Sequences

`Sequence` is an interesting object. It is the head of expressions that match patterns like __ (double blank) and ___ (triple blank).

```
ClearAll[f, g]
f[a, b, c] /. f[x__] -> x
Sequence[a, b, c]
```

You seldom see the head `Sequence` because it disappears as soon as it is wrapped in any other expression. This is called *sequence splicing:*

```
g[x, %, y]
g[x, a, b, c, y]
```

Note that sequence splicing happens even when the `Sequence` is inside of a head having the `HoldFirst`, `HoldRest`, or `HoldAll` attribute. This behavior is implied by step 9 of the main evaluation loop.

```
Hold[Sequence[a, b, c, d]]
Hold[a, b, c, d]
```

But a careful study of the evaluation algorithm suggests that the following construct can prevent sequence splicing from taking place:

```
Hold[f[Sequence[a, b, c, d]]]
Hold[f[Sequence[a, b, c, d]]]
```

Since the subexpression with head `f` is not evaluated, the `Sequence` is not spliced.

(Version 2.2 users note: The previous example does not work in versions of *Mathematica* prior to 3.0. However, the following alternative does work:

```
Trace[Hold[f[g[a, b, c, d]]] /. g->Sequence]
{Hold[f[g[a, b, c, d]]] /. g -> Sequence,
  Hold[f[Sequence[a, b, c, d]]]}
```

The most plausible explanation for this behavior is that the parser splices the `Sequence` before the main evaluation loop even gets a look at the input.)

All very unusual, but what is the practical use of `Sequence`? For one thing, applying `Sequence` to a nested expression is an easy way to obliterate the head of that expression. For example, suppose we were given a list of integers that we wanted to use as a single, multidimensional subscript into a nested list (see the Huffman coding example of Section 5.3.5 for a practical application). Simply passing the list of integers to `Part` would not have the desired effect:

Each element of `subs` is used as a separate subscript.

```
A = Array[a, {3, 3}];
subs = {2, 1};
A[[subs]]
{{a[2, 1], a[2, 2], a[2, 3]},
  {a[1, 1], a[1, 2], a[1, 3]}}
```

We need to turn an expression like `A[[{2, 1}]]` into `A[[2, 1]]`. Version 3.0 defines a new function, `Extract`, that solves this problem. In versions prior to 3.0, one could define `Extract` like this:

```
Extract[list_, {indices__}] := list[[indices]]
Extract[A, {2, 1}]
a[2, 1]
```

It is surprisingly difficult to achieve this effect without the use of an auxiliary function. Here is a solution using pure functions:

`##` represents all of the arguments to a pure function.

```
A[[##]]& @@ subs
a[2, 1]
```

The crux of what we are trying to accomplish is the removal of a `List` head from a nested subexpression. Therefore, `Sequence` is the right tool for the job:

```
A[[Sequence @@ subs]]
a[2, 1]
```

Another application of `Sequence` lies in passing options from one function to another, since many functions that accept options expect a sequence of them, rather than a list. The standard package `Utilities`FilterOptions`` defines a function that takes one sequence of options as an argument and returns another sequence of options. We'll see how to use this function in Section 9.2.3, "Filtering options."

7.2 Nonstandard Evaluation

Normally, the head and the parts of an expression are evaluated before any rules are applied to that expression; this is called *standard* evaluation. However, certain attributes and heads affect this process by preventing or forcing the evaluation of cer-

tain parts of an expression. This is collectively termed *nonstandard* evaluation. A good understanding of nonstandard evaluation is a prerequisite for doing any nontrivial manipulation of symbolic expressions.

The basic idea of nonstandard evaluation is not hard to understand; what makes nonstandard evaluation difficult is the myriad of special cases. Aside from the `Hold-` family of attributes and the `Evaluate` head, there are the heads `Hold`, `HoldComplete`,[3] `HoldForm`, `HoldPattern`,[4] and `Unevaluated`, which seem similar yet are subtly different.

We'll begin our study of nonstandard evaluation by reviewing the `Hold-` attributes. Then we'll go through the laundry list of special heads from the preceding paragraph. Finally, in Section 7.3 we'll concentrate on how to use these heads to operate on held expressions, i.e., to manipulate the structure of expressions without allowing them to evaluate.

7.2.1 The Hold- attributes

The attributes `HoldFirst`, `HoldRest`, and `HoldAll` were first introduced in Section 4.4, "Parameter-Passing Semantics." Basically, these attributes prevent the evaluation of the first, all but the first, or all of the arguments to a function. For example, here is how you could write a function that increments the value stored in a symbol:

```
SetAttributes[inc, HoldFirst];
inc[x_Symbol] := x = x + 1
```

This call is equivalent to `y = y + 1`.

```
y = 7;
inc[y];
y
8
```

Note that if `inc` didn't hold its argument, `y` would evaluate to `7` before the evaluation of the function, resulting in the nonsensical expression `7 = 7 + 1`.

One might well wonder how it is possible for an expression like `x = x + 1` to turn into `y = y + 1` without evaluating `x` and, in the process of doing so, `y`. This brings up a subtle point that is worth stressing: Pattern variables like `x` do not *evaluate*, in the technical sense, to the values matched by them. When a rule is matched (and as we saw in Section 7.1.1, a function definition is just a special kind of rule), *Mathematica textually substitutes* the values that are matched by the pattern variables into the corresponding placeholder symbols (i.e., `x` in the example above) in the body of the rule. If a function evaluates its arguments, the arguments evaluate before the substitution is per-

3. Version 3.0 only.
4. In versions prior to 3.0, `HoldPattern` was called `Literal`. Note that `Literal` is still supported in version 3.0 as well.

formed; otherwise they do not. (Review the evaluation algorithm in Section 7.1.3.) As a simple example of this process, consider:

```
SetAttributes[f, HoldAll]
f[x_] := g[x]
Trace[f[2 + 2]]
{f[2 + 2], g[2 + 2], {2 + 2, 4}, g[4]}
```

Note that the unevaluated sum `2 + 2` is substituted for `x` in `g[x]`. The sum subsequently evaluates (because g does *not* hold its argument), but the crucial point is that the evaluation does not occur until *after* the sum is deposited inside the g head. Had g been a function that held its argument, g would have "seen" the sum in its unevaluated form.

As another example of using `Hold-` attributes, the built-in looping constructs all hold their arguments, to prevent the loop "body" from evaluating until the loop condition can be checked.

```
Attributes[While]
{HoldAll, Protected}
```

Suppose that we wanted to implement a looping construct that had the semantics of the `do...while` loop of C or the `repeat...until` loop of Pascal (i.e., the loop condition is evaluated at the end of each loop iteration, rather than at the beginning). Here is one way to do it:

```
Attributes[doWhile] = {HoldAll};
doWhile[body_, test_] := (
    body;
    While[test, body]
)
```

Here is an illustration of the operation of `doWhile`.

```
x = 0;
doWhile[Print["body (", x, ")"]; x++,
        Print["test (", x, ")"]; x < 2]
body (0)
test (1)
body (1)
test (2)
```

Note that the arguments to `doWhile` are passed into `While` without being evaluated. The call to `doWhile` shown above evaluates to the following intermediate form:

```
Print["body (", x, ")"]; x++;
While[Print["test (", x, ")"]; x < 2,
        Print["body (", x, ")"]; x++]
```

All of the built-in functions with Hold- attributes appear in Tables 7-3, 7-4, and 7-5 at the end of this chapter. Let's examine a few of these functions and see what the rationale is behind the choice of attribute.

Set (=) is an example of a function with the HoldFirst attribute. Thus, in an expression like y = z (the internal form of which is Set[y, z]), z is evaluated but y is not. It's easy to see why the first argument to Set shouldn't be evaluated: The purpose of Set is to assign a value to y, not to what y might evaluate to. For example,

This assignment is equivalent to y = 3.

```
y = 2; z = 3;
y = z;
y
3
```

If y had been evaluated, the expression y = z would have turned into 2 = 3, which is nonsense.

In contrast to Set, SetDelayed (:=) has the HoldAll attribute. SetDelayed most commonly is used to define functions so as to prevent the evaluation of any symbols in the body of the function that already have values. The differences between Set and SetDelayed, and when to use each, have already been discussed in detail (Section 4.1.5, "Set versus SetDelayed").

RuleDelayed (:>) is an example of the rare breed of functions that have the HoldRest attribute. All rules evaluate their first part (the left-hand side of the rule), since they almost always are applied to fully evaluated expressions. However, it sometimes is necessary to prevent the evaluation of the right-hand side of a rule, for all of the same reasons that one might wish to prevent the evaluation of the right-hand side of a function definition. RuleDelayed solves this problem by evaluating its first part but not evaluating its second part. Thus, it needs the HoldRest attribute.

We will have a bit more to say about Set, SetDelayed, Rule, and RuleDelayed when we discuss HoldPattern in Section 7.2.5.

Version 3.0 adds a few new Hold- type attributes: SequenceHold and HoldAllComplete. As its name implies, SequenceHold prevents sequence splicing.

Note that SequenceHold does not prevent any parts from evaluating.

```
SetAttributes[sh, SequenceHold]
sh[w, Sequence[x, y], z]
sh[w, Sequence[2, 3], 3]
```

HoldAllComplete, on the other hand, prevents *any* modification or evaluation of the parts of an expression, even when those parts have special heads such as Sequence or Evaluate (discussed next). Such a heavy-handed approach is necessary only in very unusual circumstances, such as trying to define formatting rules for expressions involving special heads. We'll see examples that make use of HoldAllComplete in Sections 7.3.3 and 7.3.7.

7.2.2 Evaluate

`Evaluate` is a head that can be used to force an argument to a function to evaluate, even if that argument is in a held position. It frequently is applied to arguments of numeric functions that normally hold their arguments, to allow useful symbolic simplifications to take place prior to numerical evaluation.

For example, the `Plot` command does not evaluate its arguments. This is done under the assumption that evaluating the arguments symbolically might not be a correct thing to do (for example, if the argument to be plotted contained conditional execution constructs such as `If`). Instead, the expression to be plotted is evaluated numerically for each plot point chosen by `Plot`. Sometimes this strategy backfires, however: It is easy to construct cases in which it is more efficient to evaluate the expression before plotting, rather than every time a value is plugged in. Here's an example of the time taken to compute, but not render, a plot of the function given by the first 20 terms of the exponential expansion:

The `DisplayFunction->Identity` option suppresses rendering of the image.

```
Plot[Sum[x^k/k!, {k, 1, 20}], {x, 0, 1},
     DisplayFunction->Identity] // Timing
{1. Second, -Graphics-}
```

The computation can be sped up tremendously by telling the kernel that the first argument *should* be evaluated.

```
Plot[Evaluate[Sum[x^k/k!, {k, 1, 20}]], {x, 0, 1},
     DisplayFunction->Identity] // Timing
{0.0666667 Second, -Graphics-}
```

In the first case, the `Sum` expression evaluates to `1 + x + x^2/2 + ...` every time `Plot` attempts to numerically evaluate it. In the second case, the symbolic transformation occurs only once, before `Plot` begins its work.

Another common use of `Evaluate` with `Plot` involves plotting a table of functions. That example is covered by many different authors (including *The Mathematica Book*, §1.9.1) and so it will not be repeated here.

For a rather unorthodox use of `Evaluate`, we can force the left-hand side of a `Set` operation to evaluate to simulate "pointer dereferencing":

`a` behaves as a "pointer" to `b`.

```
Clear[a, b]
a = b; Evaluate[a] = 2;
{OwnValues[a], OwnValues[b]}
{{HoldPattern[a] :> b}, {HoldPattern[b] :> 2}}
```

Don't try to carry this analogy too far, though! It's very difficult to implement true pointer semantics in *Mathematica* because we can't make expressions evaluate only *part* of the way. A slight modification to the previous example demonstrates the difficulty:

There's no way to set b using this technique, because a evaluates all the way to c before the Set is performed.

```
Clear[a, b, c]
a = b; b = c; Evaluate[a] = 2;
{OwnValues[a], OwnValues[b], OwnValues[c]}
{{HoldPattern[a] :> b}, {HoldPattern[b] :> c},
  {HoldPattern[c] :> 2}}
```

Finally, note that in order for `Evaluate` to have an effect, the expression of which it is a part must be evaluated. In the next example, since the subexpression with head `f` is not evaluated, `f`'s arguments are not checked; `Evaluate` goes unnoticed.

```
Hold[f[Evaluate[1 + 1]]]
Hold[f[Evaluate[1 + 1]]]
```

7.2.3 Hold, HoldForm, and HoldComplete

`Hold` and `HoldForm` are wrappers that prevent expressions from evaluating. There's nothing very special about them; they simply are symbols that have the `HoldAll` attribute. The only difference between them is that, in standard output format, the head `HoldForm` does not appear.

The `HoldForm` head "vanishes," leaving an impossible-looking result.

```
{Hold[1 + 1], HoldForm[1 + 1]}
{Hold[1 + 1], 1 + 1}
```

The head is still there, however.

```
InputForm[%]
{Hold[1 + 1], HoldForm[1 + 1]}
```

`HoldForm` is used by the `Trace` function, for example, in order to return partially evaluated expressions without undesirable visual clutter. Every element of a trace is wrapped in `HoldForm`:

```
Trace[1 + 2 * 3] // InputForm
{{HoldForm[2 * 3], HoldForm[6]}, HoldForm[1 + 6],
  HoldForm[7]}
```

Note that arguments to `Hold` or `HoldForm` that have the head `Evaluate` will still get evaluated. Not so with `HoldComplete`, which is analogous to `Hold` except that it has the `HoldAllComplete` attribute:


```
HoldComplete[Evaluate[1 + 1], Sequence[a, b]]
HoldComplete[Evaluate[1 + 1], Sequence[a, b]]
```

There are sneaky ways to cause evaluation of the parts of an expression with a `Hold` or `HoldForm` head (or any head having the `HoldAll`, `HoldFirst`, or `HoldRest` attribute). Like any other expression, these expressions are checked not only for downvalues, but also for upvalues! That makes bizarre behavior like the following possible:

f and g are given some unusual upvalues, and f evaluates to g.

```
ClearAll[f, g]
Hold[f] ^:= "what"
Hold[g] ^:= "the heck?"
f = g;
```

Since f does not evaluate, the upvalue for Hold[f] is used.

```
Hold[f]
what
```

In this case, f evaluates to g, and then the upvalue for Hold[g] is used!

```
Hold[Evaluate[f]]
the heck?
```

Note that the behavior just illustrated cannot occur with HoldComplete, because HoldComplete prevents a search for upvalues. However, if you define downvalues for HoldComplete (after unprotecting it first), those downvalues will be applied.

7.2.4 Unevaluated

Unevaluated is a special head that can be wrapped around an argument to any function to prevent the evaluation of that argument before the function is called. The beauty of Unevaluated is that it is invisible to the function being called — it makes the function temporarily behave as though it had one of the Hold- attributes.

x is a symbol that evaluates to 7.

```
x = 7;
```

Head does not hold its argument, so Head[x] returns Head[7].

```
Head[x]
Integer
```

Unevaluated allows us to get the head of x.

```
Head[Unevaluated[x]]
Symbol
```

Unevaluated may seem similar to Hold, but there is an important difference: Hold (or HoldForm) is visible to the function being called.

```
Head[Hold[x]]
Hold
```

As we can see from steps 7 and 16 of the main evaluation loop (Section 7.1.3), the special behavior of Unevaluated is accomplished by removing the head Unevaluated from an expression before rules for the surrounding expression are applied, and restoring the head to the expression afterward if no applicable rules were found.

This function returns its argument, if the argument is an integer.

```
ClearAll[f]
f[y_Integer] := y
```

x evaluates to 7 before pattern matching, so the rule applies.

```
Trace[f[x]]
{{x, 7}, f[7], 7}
```

Here, x does not evaluate, the rule does not apply, and the head Unevaluated is restored.

```
Trace[f[Unevaluated[x]]]
{f[x], f[Unevaluated[x]]}
```

There is one particular case in which the treatment of Unevaluated is particularly perplexing:

Why is Unevaluated still there?

```
FullForm[Unevaluated[x]]
Unevaluated[x]
```

The explanation for this behavior is that FullForm doesn't do anything with its argument, and the output you see shouldn't be considered its "return value." Rather, FullForm is simply a wrapper that alerts the output routines that the expression inside it should be formatted in a special way. Since output formatting happens after the evaluation process is over, the head Unevaluated already has been restored.

In fact, the head FullForm is still wrapped around the output, but it doesn't print. We can verify this as follows:

```
Head[FullForm[Unevaluated[x]]]
FullForm
```

We'll have more to say about output formatting in Section 9.4, "Custom Output Formats."

7.2.5 HoldPattern (a.k.a. Literal)

HoldPattern (which was known as Literal in versions prior to 3.0) is another symbol with the HoldAll attribute, but unlike Hold or HoldAll, and like Evaluate or Unevaluated, it gets special treatment by the kernel. You might have noted, however, that nowhere does this head appear in the description of the main evaluation loop in Section 7.1.3. That is because HoldPattern is recognized by the pattern matcher.

We have run across HoldPattern several times so far, generally when we have inspected the rules associated with a symbol. For example,

```
ClearAll[f]
f[x_] := x^2
DownValues[f]

{HoldPattern[f[x_]] :> x^2}
```

The purpose of the head HoldPattern is to keep the expression f[x_] on the left-hand side of the downvalue from evaluating to (x_)^2. However, unlike other heads

that hold their contents, `HoldPattern` does not affect pattern matching. Thus, the rules in `Downvalues[f]` can be used as they are, without any special processing.

Occasionally you may need to use an explicit `HoldPattern` on the left-hand side of a `Rule` or `RuleDelayed`. This typically is necessary only when the expression you are trying to match is inside of a held expression. For example, the following doesn't work as expected because the left-hand side of the rule evaluates before `ReplaceAll` tries to use the rule.

```
Hold[3 * 3] /. p_ * p_ -> p^2
Hold[3 3]
```

A trace shows that `p_*p_` evaluates to `(p_)^2` before pattern matching occurs.

```
Trace[Hold[3 * 3] /. p_ * p_ -> p^2]
{{{(p_) (p_), (p_)^2}, (p_)^2 -> p^2, (p_)^2 -> p^2},
  Hold[3 3] /. ((p_)^2 -> p^2), Hold[3 3]}
```

This behavior is surprising only the first time you see it — after all, `p_` is simply an expression (`Pattern[p, Blank[]]`), and any expression times itself is equal to the expression squared. To get around this difficulty, simply enclose the left-hand side of the rule in `HoldPattern`:

```
Hold[3 * 3] /. HoldPattern[p_ * p_] -> p^2
Hold[3^2]
```

In a sense, then, `HoldPattern[`*`lhs`*`] -> `*`rhs`* is complementary to *`lhs`* `:>` *`rhs`* (the former holds the left-hand side of the rule but evaluates the right-hand side). And, of course, `HoldPattern[`*`lhs`*`] :>` *`rhs`* does not evaluate the expressions on either side of the rule.

A very rare use for `HoldPattern` is to prevent the evaluation of *individual parts* of the left-hand side of a rule or function definition. Recall that in Section 7.2.1 we discussed the rationale for the choice of attributes of functions like `Set`, `SetDelayed`, `Rule`, and `RuleDelayed`. To summarize, the `Rule-` functions generally are applied expressions that already have evaluated, hence their left-hand sides are evaluated; the `Set-` functions, in contrast, are used to create rules for expressions that have *not* yet evaluated, hence their left-hand sides are held. This is a slight oversimplification, however. The real truth is, the *parts* of the expression on the left-hand side of a `Set-` function *are* evaluated, because pattern matching is done after the *parts* of an expression have been evaluated. For example:

A definition like this ...

```
ClearAll[a, b, f, g]
{a, b, f} = {1, 2, g};
f[a, b] := "strange"
```

does not create a rule for `f[a, b]`.

```
DownValues[f]
{}
```

Rather, it creates a rule for `g[1, 2]`.

```
DownValues[g]
{HoldPattern[g[1, 2]] :> strange}
```

In those rare cases in which you want some of the parts on the left-hand side of a `Set-` function to evaluate but not others, you can wrap the individual parts that you *don't* want evaluated in `HoldPattern`.

This creates a rule for `g[a, 2]`.

```
Clear[g]
f[HoldPattern[a], b] := "stranger"
DownValues[g]
{HoldPattern[g[HoldPattern[a], 2]] :> stranger}
```

Note that if g had been given the `HoldFirst` attribute *before* this definition were made, then no `HoldPattern` would have been required.

```
Clear[g]
SetAttributes[g, HoldFirst];
f[a, b] := "strangest"
DownValues[g]
{HoldPattern[g[a, 2]] :> strangest}
```

The general rule when making definitions is this: any part at level 1 of a definition that is matched by a held argument is not evaluated, and all other parts at level 1 (including the head) are evaluated. However, rules associated with the head of the definition are not applied. This may sound complicated, but it meshes perfectly with the way that expressions are evaluated, as the following example demonstrates:

You should trace this evaluation if you don't understand it completely.

```
f[a, b]
strangest
```

7.2.6 Verbatim

`Verbatim`, which is new to version 3.0, is a "quoting" mechanism for patterns. Like `HoldPattern`, it is recognized by the pattern matcher. However, `Verbatim` has a different purpose: It specifies that the expression it contains must be matched exactly, without any special interpretation by the pattern matcher. For example, ordinarily a pattern like `x_` matches any expression. When wrapped in `Verbatim`, however, it matches only the expression `x_`:

```
Clear[x, y]
{x, x_, y, y_} /. Verbatim[x_] -> matched
{x, matched, y, y_}
```

Suppose that we wanted to write a pattern that could match any pattern variable in an expression, rather than a specific pattern variable such as `x_`. The canonical internal representation of any pattern variable is `Pattern[`*`var`*`, `*`expr`*`]`:

```
FullForm[{x_, y__, z:j[_Integer]}]
List[Pattern[x, Blank[]],
  Pattern[y, BlankSequence[]],
  Pattern[z, j[Blank[Integer]]]]
```

One way to accomplish this task is as follows:

```
{x, x_, y, y_} /. p_Pattern :> q[p[[1]]]
{x, q[x], y, q[y]}
```

But now suppose that we want to destructure the pattern variable with the rule, i.e., we want to match a pattern of the form `Pattern[var_, _]` directly rather than having to pick out the individual pieces using part extraction. Specifying a pattern like `Pattern[var_, _]` will result in an error:

```
{x, x_, y, y_} /. Pattern[var_, _] -> q[var]
Pattern::patsym:
   First element in pattern Pattern[var_, _]
     is not a symbol.
{x, x_, y, y_}
```

The problem is that the pattern matcher sees a `Pattern` object and assumes that it represents an ordinary pattern variable, which must have a plain symbol as its first part. The solution to this problem is to use `Verbatim` to prevent the `Pattern` object from being recognized as such by the pattern matcher.

```
{x, x_, y, y_} /. Verbatim[Pattern][var_, _] -> q[var]
{x, q[x], y, q[y]}
```

Note that `HoldPattern` would also do the job in this case, but conceptually, `HoldPattern` is the wrong tool — the objective is not to prevent evaluation, but rather to prevent the special interpretation of the symbol `Pattern`.

7.2.7 Magic cookies

The symbols `Evaluate`, `Unevaluated`, and `Sequence` are *magic cookies*. This is a whimsical term used by computer scientists to refer to any type of value that has special significance to the software system of which it is a part. The behavior of these symbols is not a result of any values or attributes that they possess; rather, it is "wired into" the kernel. This implies that these behaviors simply can't be altered, nor can they be duplicated.

For example, there is no way to give any other symbol the special properties of these symbols:

```
z = Evaluate;
Hold[z[2 + 2]]
Hold[z[2 + 2]]
```

```
z = Unevaluated;
Head[z[2 + 2]]
Unevaluated

z = Sequence;
Hold[a, z[b, c], d]
Hold[a, z[b, c], d]
```

Likewise, there is no way to take away their special properties:

```
Block[{Evaluate = this is futile},
    Hold[Evaluate[2 + 2]]
]
Hold[4]

Block[{Unevaluated = another futile gesture},
    Head[Unevaluated[2 + 2]]
]
Plus
```

(*Don't try this at home:* For reasons that are much too involved to explain here, *never* try to assign a value to `Sequence`. Doing so may cause the link between the front-end and the kernel to break, thus terminating the kernel!)

All of these mysterious results can be explained by studying the main evaluation loop algorithm. The point of this note is that one should not think of these *magic cookies* as ordinary symbols; they cannot be forged, nor can their special properties be deactivated.

7.3 Working with Held Expressions

It may seem strange to discuss the topic of working with held expressions in a chapter entitled "Expression Evaluation," since this topic is all about expression *non*evaluation! However, this material logically belongs here since a thorough understanding of the evaluation process — which you of course possess at this point — is a prerequisite for working with held expressions.

7.3.1 How do held expressions arise?

It is anticipated that you may well be wondering how held expressions arise in practice. After all, the examples in the previous section all seem very contrived. Before proceeding, then, we will motivate the discussion.

The most common source of held expressions is an argument passed to a function with a `Hold-` attribute. Up until now, all of our examples of such functions have immediately "passed the buck" to system-defined functions — for example, the `inc` and

`doWhile` functions in Section 7.2.1 simply pass their arguments to `Set` and `While`, respectively, which do the real work. If you want to write any nontrivial `Hold`-type functions, you have to learn to deal with held expressions. And whether or not you *want* to write such a function is largely irrelevant — for example, if you need to override a built-in function that holds its arguments, your definitions will need to hold the arguments in the same way.

Another source of held expressions is input in the form of character strings, whether from the user (the `InputString` function prompts the user for a string), the kernel (the `Names` function is a prime example — see Section 7.3.5), or a file (see Chapter 12, "Input/Output"). The function `ToHeldExpression` converts a character string into an expression that is immediately wrapped in `Hold`.

```
ToHeldExpression["1/0"]
```

```
Hold[1/0]
```

(Note that in version 3.0, `ToExpression` takes an optional parameter that consists of an arbitrary head to wrap around the converted expression, thus making `ToHeldExpression` obsolete.)

Still other sources of held expressions are the own-, down-, and upvalues of symbols. Later in this chapter we will write a function that examines these expressions to determine the dependencies that exist between various symbols.

Finally, functions such as `Trace` return values that are chock-full of held expressions. In fact, `Trace` has a relative, `TraceScan` (Section 13.2.4), that allows you to supply a function that is applied to every element of a trace as the evaluation unfolds. Obviously, the argument to such a function must not be allowed to evaluate.

Once you have a held expression, what can you do with it? Quite a bit, as it turns out. The next few sections will cover the various functions and techniques that can be used on held expressions and some practical examples that exploit those techniques.

7.3.2 Part extraction and replacement

The most important thing to realize about held expressions is that they are, first and foremost, expressions. Their parts and levels are numbered just as any other expression's are, and you can extract and replace those parts.

```
a = 3; b = 4;
```

Note that `Unevaluated` would also work here in place of `HoldPattern`.

```
Position[Hold[a + b], HoldPattern[b]]
```

```
{{1, 2}}
```

Most likely, a part extracted from a held expression may be something that shouldn't be allowed to evaluate, as the following example shows:

b evaluates as soon as it is outside of the Hold.

```
Trace[Hold[a + b][[1, 2]]]
{Hold[a + b][[1, 2]], b, 4}
```

You can prevent this from happening by using HeldPart instead of Part, which wraps the extracted part in Hold before returning it. The arguments to HeldPart are the same as the arguments to Part (but unlike Part, HeldPart has no special input form).

```
HeldPart[Hold[a + b], 1, 2]
Hold[b]
```

The Extract function, which is new to version 3.0, provides a more general mechanism for extracting parts of expressions. Extract[*expr*, *subscriptlist*, *head*] extracts the part of *expr* specified by *subscriptlist* and wraps the result in *head*. So the previous result could also be obtained this way:

```
Extract[Hold[a + b], {1, 2}, Hold]
Hold[b]
```

Like Part and HeldPart, Extract will extract more than one part of an expression at a time if its second argument is a list of part specifications. But Extract is more flexible than HeldPart because the head of the result doesn't have to be Hold. Furthermore, the third argument to Extract can be omitted, so Extract subsumes the functionality of Part as well (refer to the discussion of Sequence in Section 7.1.4 for an example).

You also can replace parts of a held expression with other held expressions. ReplaceHeldPart takes the same arguments as ReplacePart, except that if its second argument has the head Hold or HoldForm, the head is stripped off (but the contents are not evaluated) before the new expression is inserted.

```
ReplaceHeldPart[Hold[a + b], Hold[Sqrt[b]], {1, 2}]
Hold[a + Sqrt[b]]
```

In version 3.0, ReplacePart has been enhanced to subsume the functionality of ReplaceHeldPart (although the latter is still supported). This is accomplished by an optional fourth parameter to ReplacePart that specifies which part of the new expression to use as the replacement. The following example replaces part {1, 2} of Hold[a + b] with part 1 of Hold[Sqrt[b]].

```
ReplacePart[Hold[a + b], Hold[Sqrt[b]], {1, 2}, 1]
Hold[a + Sqrt[b]]
```

The advantage of using ReplacePart rather than ReplaceHeldPart is that the head of the second argument doesn't have to the head Hold or HoldForm, and the part of the second argument that is used as a replacement need not be part 1.

When you have massaged the held expression into exactly the form you want, you can evaluate it using ReleaseHold:

```
ReleaseHold[%]
5
```

7.3.3 Application: Modifying large lists

Although procedural programming is discouraged in *Mathematica*, sometimes there is no practical alternative. Functional programming primitives modify *copies* of their arguments; when arguments are large data structures and memory is tight, this simply is not feasible. In situations like this, there may be no alternative but to operate on the original data directly, modifying it in place.

Unfortunately, in-place modification of large lists is extremely inefficient in *Mathematica*. As an example, consider the following two ways of computing the first moving average of a list of numbers:

Here the result is computed in place (i.e., it overwrites the original data).

```
s = Range[1000];
Do[s[[i]] = (s[[i]] + s[[i + 1]])/2,
   {i, Length[s] - 1}]; // Timing
{3.01667 Second, Null}
```

Here the result is stored in a different list than the original data. The creation of the destination list is included in the timing.

```
s = Range[1000];
(t = Table[0, {1000}];
 Do[t[[i]] = (s[[i]] + s[[i + 1]])/2,
    {i, Length[s] - 1}];) // Timing
{0.633333 Second, Null}
```

What could possibly explain such a great disparity in execution time? The answer is that each time an expression like `s[[i]]` is evaluated, the list `s` is evaluated. During the evaluation of `s` the kernel has to attempt to evaluate each element of `s` and check to see if there are upvalues defined for `List[___, element, ___]`. Even if it turns out that none of the elements of the list need to be evaluated, the process of ascertaining this fact takes an amount of time that is proportional to the length of the list. Ordinarily the kernel avoids this overhead by keeping track of whether or not the entire expression has changed since the last time it underwent a full evaluation (see step 4 of the main evaluation loop in Section 7.1.3). In the second algorithm, since `s` never changes, the kernel never has to fully evaluate `s`. In the first algorithm, however, every time an element of `s` is modified, `s` is marked as modified. Thus, `s` undergoes a full evaluation on each loop iteration, which makes the running time of the first algorithm *quadratic* in the length of the list, rather than linear.

We can prevent this overhead by changing the head of `s` to `HoldComplete`. As discussed at the end of Section 7.2.3, the parts of a `HoldComplete` object (or any expression whose head has the `HoldAllComplete` attribute) are *completely* ignored — there isn't even a scan for upvalues. Nevertheless, we still are able to index into a `HoldComplete` object as we would any other expression.

```
s = HoldComplete @@ Range[1000];
Do[s[[i]] = (s[[i]] + s[[i + 1]])/2,
    {i, Length[s] - 1}]; // Timing
{0.633333 Second, Null}
```

This simple change brings the performance of the in-place algorithm up to the level of the data-copying algorithm, without requiring any extra memory!

Unfortunately this trick does not work if `Hold` is used instead of `HoldComplete` (see the exercise), which would seem to leave version 2.2 and earlier users out in the cold. However, David Withoff has suggested the following solution to this problem [Gayley 94a]: Wrap the entire list in `Hold` (as opposed to applying `Hold`) and then *double*-index the resulting expression. Because the argument to `Hold` (the original list) is not evaluated, there is no scan through the list elements and thus the time for each indexing operation is independent of the length of the list.

Note the extra index in each indexing operation. `s[[1]]` is the original list.

```
s = Hold[Evaluate[Range[1000]]];
Do[s[[1, i]] = (s[[1, i]] + s[[1, i + 1]])/2,
   {i, Length[s[[1]]] - 1}]; // Timing
{0.65 Second, Null}
```

This technique may be slightly slower than the `HoldComplete` technique because of the extra subscript in each indexing operation.

Exercise

1. Try using the `Hold` instead of `HoldComplete` in the single-indexing algorithm. Time the computation. How does it compare to the other methods, and what conclusions can you draw? (If you don't understand the result, review the pathological examples given at the end of Section 7.2.3 for some clues.)

7.3.4 Functional operations

You can also use functional operations on a held expression. Here we use `MapAt` to accomplish the same transformation for which we used part replacement in Section 7.3.2.

```
MapAt[Sqrt, Hold[a + b], {1, 2}]
Hold[a + Sqrt[b]]
```

One very useful functional operation on held expressions is plain old `Apply` (`@@`). `Apply` allows you to "hand off" a held expression to another function that operates on held expressions, *without* allowing the parts of the held expression to evaluate. For example, suppose we wanted to use the contents of the held expression above on the left-hand side of a rule. We could do this easily by applying `HoldPattern`, which replaces the head `Hold`:

Now this expression can be used for pattern matching.

```
HoldPattern @@ %
HoldPattern[a + Sqrt[b]]
```

Another example of using Apply to hand off a held expression from one function to another is the following. Suppose we have a large list, and we want to see if all of the elements in the list satisfy some predicate. For example, here we test a list of integers to see if all of them are positive:

```
z = Range[10000];
And @@ Positive /@ z // Timing
{0.5 Second, False}
```

Testing to see if all the elements are negative takes the same amount of time, even though the result could be inferred merely by testing the first element.

```
And @@ Negative /@ z // Timing
{0.466667 Second, False}
```

The problem is that `Negative /@ z` is evaluating to a list of 10,000 `False` symbols before And checks any of them. And has the `HoldAll` attribute, so we should be able to optimize this case by postponing the evaluation of the 10,000 calls to `Negative` until after And has been wrapped around them. We can do that as shown below:

```
And @@ Negative /@ Hold @@ z // Timing
{0.216667 Second, False}
```

`Negative /@ Hold @@ z` creates an expression of the form `Hold[Negative[1], Negative[2], ..., Negative[10000]]`. When And is applied to this expression, the individual parts are evaluated under the control of And, which bails out as soon as any of them evaluate to `False`. The following trace shows this explicitly:

```
Trace[And @@ Negative /@ Hold @@ Range[-2, 10000],
      Negative[_]]
{{Negative[-2]}, {Negative[-1]}, {Negative[0]}}
```

(Naturally, doing the test this way slows down the first case, i.e., when all of the elements of the list satisfy the predicate. However, the penalty is minimal compared to the saving just illustrated.)

Mapping or applying pure functions to held expressions can be tricky. To illustrate the difficulty, consider the problem of determining the head of the expression inside of a held expression:

Since Head does not hold its argument, `a + b` evaluates to an integer.

```
Trace[Head @@ Hold[a + b]]
{Head @@ Hold[a + b], Head[a + b],
  {{a, 3}, {b, 4}, 3 + 4, 7}, Head[7], Integer}
```

One might expect that this problem could be circumvented by using a pure function of the following form:

```
Head[Unevaluated[#]]& @@ Hold[a + b]
Integer
```

The problem is that `Apply` doesn't literally wrap the pure function around the contents of the held expression, as in `Head[Unevaluated[a + b]]`. Instead, the intermediate form `Head[Unevaluated[#]]&[a + b]` is created, and the argument `a + b` evaluates before the pure function does. As we saw in Section 7.1.4, the solution to this problem is to specify a `Hold-` attribute as the third argument to `Function` (note that there is no shorthand syntax for this):

This pure function has the `HoldFirst` attribute.

```
Function[x, Head[Unevaluated[x]], HoldFirst] @@
    Hold[a + b]
Plus
```

Another useful functional operation on held expressions is `Thread`, which can be used to "push" the `Hold` head one level down in the expression. For example:

The second argument to `Thread` specifies the head to be threaded over.

```
Thread[Hold[a + b], Plus]
Hold[a] + Hold[b]
```

Best of all, `Thread` can be used to reverse the process as well!

```
Thread[%, Hold]
Hold[a + b]
```

Exercise

1. Implement your own version of `Extract` by using `Part`, `Sequence`, and `Apply`. Your function should be able to wrap an arbitrary head around the extracted subexpression before that subexpression evaluates (see the `Extract` example in Section 7.3.2).

7.3.5 Application: Filtering symbols

Finding all symbols having certain properties is a good example of using functional operations on held expressions. `Names["stringpattern"]` returns a list of the names of all symbols that match the given *string pattern* — e.g., a string containing wildcard characters like `"*"` (matches zero or more characters) or `"@"` (matches one or more characters, excluding uppercase letters). For example,

There are over 1500 names defined by the version 3.0 kernel!

```
n = Names["System`*"];
Short[n]
{Abort, AbortProtect, <<1524>>, $VersionNumber}
```

The output of `Names` is a list of strings, not symbols (you can verify this using `InputForm`), because symbols might evaluate. Now we could, for example, select the names of all symbols in such a list that have a certain attribute by using a predicate like

`MemberQ[Attributes[#], attrib]&`. For example, here are the names of all system-defined symbols that have the `HoldRest` attribute:

```
Select[n, MemberQ[Attributes[#], HoldRest]&]
{If, PatternTest, RuleDelayed, Save}
```

This technique was used to generate Tables 7-3, 7-4, and 7-5 at the end of this chapter, as well as Table 3-1 on page 67.

This strategy is fine for testing attributes, because `Attributes` accepts a string as an argument. However, functions such as `Options`, `MessageName` (`"::"`), `OwnValues`, etc., are not so accommodating — they require actual symbols as arguments. Converting all of the names to symbols using `ToExpression` is problematic, because some symbols might evaluate, rendering them unavailable for further analysis:

Note that `ToExpression` is `Listable`.

```
ToExpression[n];
Take[%, {1510, 1520}]
{{Graphics, Graphics3D, ContourGraphics,
   DensityGraphics, SurfaceGraphics, GraphicsArray,
   Sound}, $SyntaxHandler, Power Macintosh,
  MacintoshRoman, PowerMac, MathTemp., {}, 1/60,
  Gracilis:Math:Mathematica 3.0:, None, None}
```

The solution is to use `ToHeldExpression` instead:

```
hn = ToHeldExpression[n];
Take[hn, {1510, 1520}]
{Hold[$SuppressInputFormHeads], Hold[$SyntaxHandler],
  Hold[$System], Hold[$SystemCharacterEncoding],
  Hold[$SystemID], Hold[$TemporaryPrefix],
  Hold[$TextStyle], Hold[$TimeUnit],
  Hold[$TopDirectory], Hold[$TraceOff],
  Hold[$TraceOn]}
```

All of those `Hold` heads are going to get in the way. It would be nice to get rid of them, but then we have the same problem with uncontrolled evaluation as before! Obviously, we must hold the entire list *before* attempting to remove the `Hold` heads from the individual symbols.

```
hhn = Hold @@ hn;
Short[hhn]
Hold[Hold[Abort], <<1525>>, Hold[$VersionNumber]]
```

Now we can exploit the fact that `Flatten` can flatten nested expressions having any head, not just lists:

The extra arguments to `Flatten` are a level specification and the head to flatten.

```
hsyms = Flatten[hhn, 1, Hold]
Short[hsyms];
Hold[Abort, A<<9>>ct, <<1524>>, $VersionNumber]
```

The elements of `hsyms` are symbols, not just names of symbols. The fact that the head of the expression is `Hold` rather than `List` does not prevent us from using `Map`, `Select`, or any other functional operation on it.

Now suppose that we want to select those symbols that have the option `Heads`. This is equivalent to testing if the pattern `Heads->_` is a member of `Options[`*`sym`*`]`. Since `Options` does not hold its argument, the form we need to use is `Options[Unevaluated[`*`sym`*`]]`. Also note that we need to make the pure function hold its argument (try doing this without the attribute, to see the difference).

This technique was used to generate candidates for Table 5-1 on page 106.

```
Select[hsyms,
    Function[x,
        MemberQ[Options[Unevaluated[x]], Heads->_],
        HoldFirst]
]
Hold[Apply, Cases, Count, DeleteCases, FreeQ, Level,
  Map, MapAll, MapAt, MapIndexed, MapThread, MemberQ,
  Position, Scan]
```

Here is a slightly more complicated example that finds all system-defined symbols with the word `pattern` in their usage message. Note that not all system-defined symbols have usage messages, so `StringQ[x::usage]` is used to check for the existence of a usage message before attempting to use `StringMatchQ`.

This technique was used to generate candidates for Table 5-2 on page 106. This evaluation may take a long time.

```
Select[hsyms,
    Function[x,
        StringQ[x::usage] &&
            StringMatchQ[x::usage, "*pattern*"],
        HoldFirst
    ]
]
Hold[Alternatives, Blank, BlankNullSequence,
  BlankSequence, Cases, Clear, Condition, Count,
  Default, DeleteCases, ExcludedForms, Expand,
  ExpandAll, ExpandDenominator, ExpandNumerator,
  FileNames, Flat, LinkPatterns, ThisLink,
  $CurrentLink]
```

Exercises

1. The `Thread` trick from Section 7.3.4 is *almost* the right tool for converting `hn` to `hsyms`. The problem is that the result is an expression of the form `Hold[{`*`sym`*`, ...}]` rather than `Hold[`*`sym`*`, ...]`. Use `Apply` and `Sequence` to obliterate the extra level in this expression.

2. Find all system-defined symbols that have format values.
3. Find all system-defined symbols that have anything to do with graphics.

7.3.6 Rule substitution

You can use rule substitution on held expressions, although frequently when you do so, it is necessary to wrap part or all of the left-hand side of the rule in `HoldPattern`, as explained in Section 7.2.5. For example, here is yet another way to turn `Hold[a + b]` into `Hold[a + Sqrt[b]]`:

```
Hold[a + b] /. HoldPattern[b] :> Sqrt[b]
Hold[a + Sqrt[b]]
```

`HoldPattern` keeps the left-hand side of the rule from evaluating, and `RuleDelayed` keeps the right-hand side from evaluating.

One of the most powerful uses of pattern matching in *Mathematica* is *destructuring*, which is the process of using patterns to take apart expressions. Here's an alternative way to solve the problem, posed in the last section, of finding all built-in symbols having the `Heads` option:

```
n = Names["System`*"];
Cases[ToHeldExpression[n], Hold[sym_] /;
        MemberQ[Options[Unevaluated[sym]], Heads->_]]
{Hold[Apply], Hold[Cases], Hold[Count],
  Hold[DeleteCases], Hold[FreeQ], Hold[Level],
  Hold[Map], Hold[MapAll], Hold[MapAt],
  Hold[MapIndexed], Hold[MapThread], Hold[MemberQ],
  Hold[Position], Hold[Scan]}
```

This technique — extracting part of a held expression and plugging it directly into another expression — is arguably more straightforward than using `Select` with a pure function with a `Hold-` attribute. The output is a bit more awkward than before, but that is easily fixed by using the `Thread` trick:

```
Thread[%, Hold]
Hold[{Apply, Cases, Count, DeleteCases, FreeQ, Level,
   Map, MapAll, MapAt, MapIndexed, MapThread,
   MemberQ, Position, Scan}]
```

For our final example of using rule substitution on held expressions, we return to the optimal matrix-chain multiplication problem of Section 6.4.2. Recall that we had constructed the following expression (page 173):

```
multorder = {{1, {2, 3}}, {{4, 5}, 6}};
```

which we needed to transmogrify into the following form (A was the list of matrices to be multiplied):

```
Dot[Dot[A[[1]], Dot[A[[2]], A[[3]]]],
    Dot[Dot[A[[4]], A[[5]]], A[[6]]]]
```

The problem, you may recall, was that a transformation like `multorder /. List->Dot` would destroy the nesting of `multorder`, since `Dot` is `Flat`. On the other hand, substituting in the matrices *before* changing the `List` heads to `Dot` would be disastrous, because the matrices themselves are nested lists! The solution we adopted at that time was to change the `List` heads in `multorder` to some new head; then we were able to replace this new head with `Dot` *after* the matrices had been inserted.

However, by holding `multorder` we can operate directly with `Dot` and avoid the need to introduce a temporary head.

```
Hold[Evaluate[multorder]] /. List->Dot
Hold[1 . 2 . 3 . 4 . 5 . 6]
```

Although it looks as though the `Dot` operators have flattened out, in fact that is simply how the output routine prints them. We can verify that the structure is still there by examining the `FullForm` of this expression:

```
FullForm[%]
Hold[Dot[Dot[1, Dot[2, 3]], Dot[Dot[4, 5], 6]]]
```

Next, we turn each index *i* into `A[[i]]` using a straightforward delayed-rule substitution:

```
% /. i_Integer :> A[[i]] // FullForm
Hold[Dot[Dot[Part[A, 1],
    Dot[Part[A, 2], Part[A, 3]]],
   Dot[Dot[Part[A, 4], Part[A, 5]], Part[A, 6]]]]
```

All that remains is to apply `ReleaseHold` to this "parenthesized" chain to evaluate all of the `Dot` operators. Because evaluation proceeds depth-first (i.e., more deeply nested dot products will be evaluated before less deeply nested ones) the parenthesization will be observed.

7.3.7 Application: Dependency analysis

Suppose that we want to analyze an expression to find all of the symbols upon which it depends. Let us take as an initial example the expression `Function[{x, y}, x + y][{u, v}]`. Note that a prerequisite for success is that this expression not be allowed to evaluate while it is being analyzed.

As a first step toward our goal, we might use `Cases` to find all of the symbols in the expression:

```
Cases[Unevaluated[Function[{x, y}, x + y][u, v]],
    s_Symbol -> HoldForm[s], {-1}, Heads->True]
{Function, List, x, y, Plus, x, y, u, v}

syms = Union[%]
{Function, List, Plus, u, v, x, y}
```

Here, `Unevaluated` prevents the expression from evaluating before `Cases` can examine it; the rule extracts each symbol and stuffs it into a `HoldForm` head to prevent the symbol from evaluating; the level specification of `-1` restricts `Cases` to searching only the lowest level of the expression (since symbols have no parts, they must be at the lowest level); and the `Heads->True` option directs `Cases` to search the heads of expressions as well as their parts.

This is a good start, but it is overly conservative: It includes the symbols `x` and `y`, which are local to the pure function and have no relationship to any "global" symbols named `x` and `y`. We might try solving this problem in the following way. First, we use `Cases` to obtain a list of all of the parameters to the pure function:

```
Cases[Unevaluated[Function[{x, y}, x + y][u, v]],
    HoldPattern[Function[{vars__}, __]] :>
        Sequence @@ Thread[HoldForm[{vars}]],
    Heads->True
]
{x, y}
```

A few points worth noting about this code: First, we used `HoldPattern` to prevent `Function` from issuing an error message about the form of its first argument. Second, we used the `Thread` trick to push the `HoldForm` head inside the list `vars`. Third, the list that results was turned into a `Sequence` so that it spliced itself into the list that is returned by `Cases`.

Now that we have the list of parameters to the pure function, we can remove those symbols from the list of all symbols found in the expression.

```
syms = Complement[syms, %]
{Function, List, Plus, u, v}
```

Unfortunately, the technique shown above is rather ad hoc, and it's quite easy to make it fail. For example, if the expression had been `x + Function[{x, y}, x + y][u, v]`, this technique could not distinguish the `x` *outside* of the pure function, which is a global symbol upon which this expression depends, from the `x` *inside* the body of the pure function, which is merely a placeholder. Or consider `Function[{x}, x] + Function[{y}, x + y]`, for which this technique would be unable to determine that the symbol `x` in `x + y` is nonlocal.

Furthermore, scoping constructs, such as `Module` and `With`, and certain types of rules also introduce temporary symbols that are distinct from global symbols. Considering that these constructs can be nested arbitrarily, it's clear that a more methodical approach is required.

The solution to this problem is to write a *parser* for *Mathematica* expressions. This is not as difficult as it sounds, because the recursive structure of expressions lends itself to being analyzed by a very simple parsing method known as *recursive descent* [Aho & Ullman 77].

A recursive descent parser parses an expression in a depth-first order. For example, to parse `head[part1, part2, ...]`, it first parses `head`, then `part1`, etc. Each of these expressions may also have a head and parts, and they are parsed in the same way. By traversing the expression in this order (the same order in which the kernel does it, incidentally), a recursive descent parser can deal with arbitrarily nested expressions without getting confused. For example, to parse `Function[x, body]`, a rule is written that removes *x* from the list of global symbols returned from parsing *body*. It's irrelevant what's in *body*; the correctness of this strategy is argued inductively.

Quite obviously, the parsing routine needs to hold its argument. We choose the `HoldAllComplete` attribute because there is no conceivable case in which we would want any part of an expression to evaluate before being parsed.[5]

```
SetAttributes[parse, HoldAllComplete];
```

The parser will be structured as a collection of *Mathematica* rules, which allows us to add functionality incrementally. The most general rule simply parses the head and each of the parts recursively:

```
parse[hd_[parts___]] :=
    Union[Flatten[parse /@ Unevaluated[{hd, parts}]]]
```

When a symbol is encountered, it is returned wrapped in `HoldForm`:

```
parse[s_Symbol] := HoldForm[s]
```

Everything else is ignored at this early stage of development.

```
parse[_] := {}
```

At this point we have implemented the functionality of the first `Cases` statement in this section:

```
parse[Function[{x, y}, x + y][u, v]]
{Function, List, Plus, u, v, x, y}
```

5. Users of versions prior to 3.0 should choose the `HoldFirst` or `HoldAll` attribute. Certain features of `parse`, pointed out below, will not be available.

The following trace shows the order in which the expression is traversed.

```
Trace[parse[Function[{x, y}, x + y][u, v]], parse[__]]
{parse[Function[{x, y}, x + y][u, v]],
  {{{parse[Function[{x, y}, x + y]],
     {{{parse[Function]}},
       {parse[{x, y}],
        {{{parse[List]}}, {parse[x]}, {parse[y]}}}},
       {parse[x + y],
        {{{parse[Plus]}}, {parse[x]}, {parse[y]}}}}}}\
   , {parse[u]}, {parse[v]}}}
```

The `HoldAllComplete` attribute allows the parser to work even on expressions that contain magic cookies (version 3.0 only):

```
parse[Evaluate[{a, Sequence[b, c], d}]]
{a, b, c, d, Evaluate, List, Sequence}
```

Now we will write a rule that handles pure functions. As mentioned above, all this rule needs to do is to remove the list of parameters to the pure function from the list of symbols found by parsing the body of the pure function. There are no techniques being used here that we haven't already seen several times.

```
parse[Function[{syms__}, body_]] :=
    Complement[parse[body], Thread[HoldForm[{syms}]]]
```

It really is that easy! One more simple rule is needed to handle the special case of functions of a single argument:

Reduce to a previously solved case.

```
parse[Function[sym_, body_]] :=
    parse[Function[{sym}, body]]
```

Below are some examples of increasing complexity that demonstrate the adaptability of this approach.

```
parse[Function[{x, y}, x + y][u, v]]
{Plus, u, v}

parse[x + Function[{x, y}, x + y][u, v]]
{Plus, u, v, x}

parse[Function[x, x + y + Function[y, x + y + z]]]
{Plus, y, z}
```

There are many more special cases that need to be taken into account in order to have a completely general, recursive-descent symbol dependency analyzer (see the exercises), but the foregoing discussion should be enough to give you a good idea of how it is done.

Given the existence of such a function, it is not difficult to write another function that takes a symbol as an argument and returns a list of all global symbols upon which it depends. Basically, all that's necessary is to parse the right-hand side of every rule for which the given symbol is the tag and to eliminate any formal parameters from each result. Here are some example rules for a symbol `f`:

`f` has two downvalues and one upvalue.

```
ClearAll[f, g]
f[x_] := x + y
f[x_, y_] := x + y + z
g[f[x_, y_, z_]] ^:= Sqrt[x^2 + y^2 + z^2]
```

A list of all types of values for a symbol can be constructed using `Through`, which takes a list of functions and applies each of them to an argument:

```
rules =
    Through[{OwnValues, DownValues, UpValues, SubValues,
             NValues, FormatValues}[Unevaluated[f]]
    ] // Flatten
{HoldPattern[f[x_]] :> x + y,
  HoldPattern[f[x_, y_]] :> x + y + z,
  HoldPattern[g[f[x_, y_, z_]]] :> Sqrt[x^2 + y^2 + z^2]}
```

Next, parse the right-hand side of each rule.

```
globals = Cases[rules, (_ :> rhs_) :> parse[rhs]]
{{Plus, x, y}, {Plus, x, y, z},
  {Plus, Power, Sqrt, x, y, z}}
```

We're not done; we need to remove any formal parameters that appear in the left-hand sides of rule definitions from the corresponding right-hand sides. Formal parameters are represented as pattern variables, and we saw in Section 7.2.6 that the pattern `Verbatim[Pattern][x_, _]` can be used to match any pattern variable. Therefore, the following function extracts all of the pattern variables from an expression:

`fpv` stands for "find pattern variables."

```
fpv[expr_] :=
    Cases[{expr},
          Verbatim[Pattern][x_, _] :> HoldForm[x],
          Infinity, Heads->True]
```

Now apply this function to the left-hand side of each rule in turn:

```
params = Cases[rules, (lhs_ :> _) :> fpv[lhs]]
{{x}, {x, y}, {x, y, z}}
```

Next, complement each element of `globals` by the corresponding element of `params`:

```
MapThread[Complement, {globals, params}]
{{Plus, y}, {Plus, z}, {Plus, Power, Sqrt}}
```

All of these steps can be consolidated into a single `Cases` statement, eliminating the need for the `MapThread`.

```
Cases[rules, (lhs_ :> rhs_) :>
        Complement[parse[rhs], fpv[lhs]]]
{{Plus, y}, {Plus, z}, {Plus, Power, Sqrt}}
```

Finally, remove duplicates from the result by applying `Union` to it.

```
Union @@ %
{Plus, Power, Sqrt, y, z}
```

Here is the fruit of our labors: a function called `dependson` that, given a symbol as an argument, returns a list of all global symbols upon which that symbol depends.

```
SetAttributes[dependson, HoldFirst];
dependson[sym_] :=
With[
    {rules = Flatten[Through[
        {OwnValues, DownValues, UpValues, SubValues,
         NValues, FormatValues}[Unevaluated[sym]]]]},
    Union @@
        Cases[rules, (lhs_ :> rhs_) :>
              Complement[parse[rhs], fpv[lhs]]]
]

dependson[f]
{Plus, Power, Sqrt, y, z}
```

Exercises

1. Change `dependson` so that in a definition like `f[g[x_]] := rhs`, `g` will appear in the result. (*Hint:* Parse the left-hand side of the definition.) However, `f` should not appear unless it is present on the right-hand side of the definition as well; otherwise there would be no way to distinguish recursive functions from nonrecursive ones.

The next two exercises are quite difficult, and it is not expected that every reader should be able to solve them. (Their solutions, along with other material, were the basis for a journal article [Wagner 96a]. The supplementary diskette contains the package *DependencyAnalysis.m*.)

2. Write a rule or rules to parse lexical scoping constructs such as `Module` (all comments here also apply to `With`). Your rules should work for arbitrarily nested expressions. For example, the following expression depends only on the symbols `a` and `z`, because `x` and `y` are local symbols:

```
Module[{x = With[{y = a}, y + z]}, x^2]
(a + z)^2
```

The output of `parse` applied to this expression should therefore be:

```
parse[Module[{x = With[{y = a}, y + z]}, x^2]]
{a, Plus, Power, z}
```

Note that the symbols on the left-hand sides of local variable initializations (e.g., y) do not appear in the output, but symbols on the right-hand sides of those initializations (e.g., a) do appear. Also note that there are three forms of local variable declarations that you have to handle: *sym*, *sym* = *expr*, and *sym* := *expr*.

As another example, the following expression *does* depend on x.

```
Module[{x = With[{y = a}, y + x]}, x^2]
(a + x)^2
```

The reason for this is that the x inside the body of the With is a global x. Therefore, x should appear in the result of parse:

```
parse[Module[{x = With[{y = a}, y + x]}, x^2]]
{a, Plus, Power, x}
```

It is recommended that you go through the above examples by hand before writing any code.

3. If we think of dependson as a relation in the mathematical sense, then dependson is *transitive:* If a dependson b and b dependson c, then a dependson c. We can define the *transitive closure* of the dependson relation, dependtrans, as follows: a dependtrans c if either a dependson c or there is some symbol b for which a dependtrans b and b dependtrans c.

   One way to compute the transitive closure of a relation is to iterate that relation on itself until a fixed point is reached. Write a function to compute dependtrans in this way. (*Hint:* Represent the relation as a list of rules of the form {HoldForm[sym1] -> dependson[sym1], HoldForm[sym2] -> dependson[sym2], ...}. This form makes it very easy to iterate the relation with a ReplaceAll operator. Use FixedPoint to carry the iteration to its conclusion.)

7.3.8 The Block trick

The construct Block[{*sym1*, *sym2*, ...}, *expr*] effectively gives the kernel a case of temporary amnesia, preventing it from applying any rules associated with any of the symbols *sym1*, *sym2*, etc. until after *expr* has been constructed and returned from the Block. This technique was first made known to the author by Allan Hayes.

The Block trick is very useful for constructing expressions that are meant to evaluate eventually, but only after they have been fully constructed. For example,

a and b do not evaluate until the Block returns.

```
Block[{a, b},
    a + b /. b -> Sqrt[b]
]
5
```

No evaluations except those involving the specified symbols are affected by the `Block`, which makes the use of this technique exceptionally straightforward.

In the context of the matrix-chain multiplication problem, here is how the `Block` trick could be used (do not attempt to evaluate this expression unless A has been properly initialized):

```
Block[{Dot},
    multorder /. {List->Dot, i_Integer:>A[[i]]}
];
```

Within the `Block`, `Dot` behaves as a symbol with no values. When `Block` returns, all of the `Dot` operations inside of the returned expression evaluate in the proper order.

Note that the `Block` trick suffers from a serious limitation: It is necessary to know in advance which symbols need to be held in order to use it.

7.4 Additional Resources

Chapter 10 of [Maeder 94a] contains many examples of applying functions to held lists of symbols.

Recursive descent parsing ([Aho & Ullman 77]) originally was developed as a method of parsing computer programming languages. It has the advantage of being easy to code by hand. Nowadays, however, more sophisticated parsers are generated by special computer programs called *parser generators*, so recursive descent usually is used only as an educational example in most texts.

Algorithms for computing the transitive closure of a relation (or equivalently, a graph) can be found in any text on algorithms and data structures. See, for example, [Cormen et al. 90].

A complete treatment of the symbol dependency analysis problem can be found in [Wagner 96a].

7.5 Appendix: Functions with Hold- Attributes

Tables 7-3, 7-4, and 7-5 on the next page show all of the functions in version 3.0 of *Mathematica* that have the `HoldFirst`, `HoldRest`, and `HoldAll` attributes, respectively. Functions marked with an asterisk (*) do not exist in earlier versions; functions marked with a dagger (†) exist but do not have the given attribute in earlier versions. In Table 7-5, certain functions with extremely long names appear at the end of the table.

Table 7-3 Functions with the `HoldFirst` attribute in version 3.0.

AddTo	AppendTo	Catch†	ClearAttributes†
Context	Debug	Decrement	DivideBy
Increment	Message	MessageName	Pattern
PreDecrement	PreIncrement	PrependTo	RuleCondition
Set	SetAttributes†	Stack	SubtractFrom
TimesBy	ToBoxForm*	Unset	UpSet

Table 7-4 Function with the `HoldRest` attribute in version 3.0.

If†	PatternTest	RuleDelayed	Save

Table 7-5 Functions with the `HoldAll` attribute in version 3.0.

AbortProtect	Alias	And	Attributes
Block	Check	CheckAbort	CheckAll*
Clear	ClearAll	Compile	CompiledFunction†
Condition	ConsoleMessage†	ConsolePrint†	ContourPlot
DefaultValues	Definition	DensityPlot	Dialog
Do	DownValues	FindMinimum	FindRoot
For	FormatValues	FullDefinition	Function
Hold	HoldForm	HoldPattern*	Information
Literal	MatchLocalNameQ	MemoryConstrained	Messages
Module	NIntegrate	NProduct	NSum
NValues	Off	On	Or
OwnValues	ParametricPlot	ParametricPlot3D	Play
Plot	Plot3D	Product	Protect
Remove	SetDelayed	StackBegin	StackComplete
StackInhibit	SubValues	Sum	Switch
Table	TagSet	TagSetDelayed	TagUnset
TimeConstrained	Timing	Trace	TraceDialog
TracePrint	TraceScan	UnAlias	Unprotect
UpSetDelayed	UpValues	ValueQ	Which
While	With	$ConditionHold	$Failed
CompoundExpression		SampledSoundFunction	

Part 3
Extending the System

8

Writing Packages

A *package* is basically a text file containing function definitions written in the *Mathematica* programming language. The purpose of a package is not to perform a particular computation, but rather to extend the system by defining new functions. In fact, *Mathematica* ships with over 100 packages, called the *standard packages* [WRI 93b]. This strategy gives users who need a particular functionality easy access to it, without forcing users who don't need it to pay the price for it (in terms of kernel memory requirements and loading time).

There is nothing magical about the standard packages; any *Mathematica* programmer can write packages of his or her own. There are, however, several conventions that packages must observe regarding the naming and visibility of symbols, proper documentation, and related issues, that are the main topic of this chapter. At the end of the chapter we also discuss the *symbol shadowing* problem, an advanced topic that may be skipped on a first reading.

8.1 Contexts

A properly designed package should not modify the definitions of any symbols that are in existence at the time the package is loaded — with the possible exception of system-defined symbols that are enhanced by the package. Obviously, the author of a package can't possibly anticipate what symbol names will be in use already. Therefore, multiple distinct symbols with the same name need to be able to coexist. *Contexts* are the mechanism used to manage such symbol names in a *Mathematica* session.

Contexts also provide *modularity:* They hide implementation details from users. Proper use of contexts in a package encourages users to access the package's data structures through a well-defined interface; this makes the package more reliable.

For these reasons, learning about contexts is a prerequisite to learning to write packages.

8.1.1 Contexts are containers for symbols

Every symbol in a *Mathematica* session belongs to some context.

System-defined symbols such as `Plus` are in the `System`` context.

```
Context[Plus]
System`
```

(The grave accent, or backquote `` ` ``, following a context name is called a *context mark*.)

Upon the first use of a new symbol name, a symbol with that name is created in whatever context happens to be the *current context*. The current context is kept in the system variable `$Context`.

```
$Context
Global`
```

Most of the time the current context is `Global``, so the vast majority of symbols that are defined by a user in the course of a *Mathematica* session fall into that context. The input shown below creates a symbol `x` in the `Global`` context:

```
x = 5;

Context[x]
Global`
```

Note that it is not necessary to assign a value to a symbol in order to create it; the mere utterance of a symbol name (metaphorically speaking, of course) causes its creation.

`y` is created simply by mentioning it.

```
Context[y]
Global`
```

Symbol names are unique within any one context, but the same symbol name appearing in different contexts refers to different symbols. How can these different symbols be specified unambiguously? Every symbol has a long name and a short name; the long name is of the form *`contextname`shortname`*.

Normally you need to use only the short name of a symbol, and symbols print as short names.

```
{x, y}
{5, y}
```

The `?` operator shows the long name of a symbol.

```
?x
Global`x
x = 5
```

The following input creates a context called `temp`, as well as a symbol `x` within that context. Note that this symbol is distinct from the symbol `Global`x`.

```
temp`x = 6;
```

You can refer to a symbol in another context by using its long name.

```
{x, temp`x}
{5, 6}
```

Or, you can change the current context to that context ...

```
Begin["temp`"]
temp`
```

and use the symbol's short name. The short name **x** now refers to **temp`x**.

```
{x, Global`x}
{6, 5}
```

This leaves the **temp`** context.

```
End[]
temp`
```

The short name **x** once again refers to **Global`x**.

```
?x
Global`x
x = 5
```

Note that **Begin** returns the name of the context that is entered, and **End** returns the name of the context that was exited. You should always use **Begin** and **End** rather than manipulating the value of **$Context** directly.

There is one important caveat regarding the use of **Begin**: Always place it on its own input line. The reason for this is that no part of an input line is evaluated until the entire line has been parsed; however, the parser uses the value of **$Context** at the time the input is read to decide how to resolve the names in the input strings to symbols. To make this more concrete, consider the following example:

```
Begin["temp`"]; Print[{x, Global`x, temp`x}]; End[];
{5, 5, 6}
```

Since the entire input line is evaluated at one time, the current context does not become **temp`** until after the parsing is completed. Therefore, the parser resolves the name **x** to the symbol **Global`x**, because **Global`** is the current context. Placing the **Begin["temp`"]** command on a separate input line gives the behavior one might expect.

```
Begin["temp`"];
Print[{x, Global`x, temp`x}]
End[];
{6, 5, 6}
```

8.1.2 Nested contexts

Contexts can be nested. For example, this input creates a symbol **x** in a subcontext **foo** of the context **temp**.

```
temp`foo`x
temp`foo`x
```

Here we evaluate `x` in three different contexts: `Global`` (the current context), `temp``, and `temp`foo``:

```
x
5

Begin["temp`"]
temp`

x
6

Begin["`foo`"]
temp`foo`

x
x
```

The `End` command undoes the action of the most recent `Begin`, thus it exits `temp`foo`` and returns to the context `temp``.

```
End[]
temp`foo`

$Context
temp`
```

Note that the specification of the subcontext name `` `foo` `` begins with a context mark. If the leading context mark is omitted, that `Begin` command creates and enters a new top-level context called `foo``.

```
Begin["foo`"]
foo`

z
z

?z
foo`z
```

The current context is now `foo``, not `temp`foo``. The function `Contexts[]`, which returns a list of all existing contexts, shows that `foo`` and `temp`foo`` are different contexts. (The other contexts in this menagerie were created as part of the kernel's initialization procedure.)

This list will be somewhat different in versions prior to 3.0.

```
Contexts[]
{Algebra`Algebraics`Private`, BoxForm`, Chase`,
  Conversion`, DSolve`, EllipFunctionsDump`, Factor`,
  FE`, foo`, Format`, Global`, Graphics`Animation`,
  HypergeometricLogDump`, Integrate`,
  Integrate`Elliptic`, Limit`, NPDSolve`, OscNInt`,
  Series`, Simplify`, Solve`, SymbolicProduct`,
  SymbolicSum`, System`, System`ComplexExpand`,
  System`CrossDump`, System`Dump`,
  System`Dump`ArgumentCount`, System`FactorDump`,
  System`Private`, temp`, temp`foo`}
```

Note that although `foo`` is not nested within `temp``, an `End` command returns to the context `temp``, since that was the current context when `Begin[temp`]` was executed.

```
End[]
foo`

$Context
temp`
```

A second `End` command is necessary to return to the `Global`` context.

```
End[]
temp`

$Context
Global`
```

There is a strong analogy between contexts and directories in a hierarchical file system. The context mark corresponds to the directory pathname separator (/ in UNIX, \ in DOS, : in MacOS), symbol names correspond to filenames, and the current context corresponds to the working directory.

However, unlike UNIX and DOS (but like the MacOS), a context mark at the beginning of a context name (as in `` `foo` ``) does not correspond to a "root" context. When the name of a context stands alone, it is an absolute name; but when it is preceded by a context mark, it is relative to the current context.

`Begin` and `End` do not correspond precisely to the UNIX `cd` or the DOS `chdir`, since they keep track of the order in which contexts are entered and left. `Begin` and `End` are more closely analogous to the UNIX shell commands `pushd` and `popd`.

8.1.3 The context search path

To allow you to use symbols from multiple contexts conveniently, *Mathematica* maintains a *context search path*, which is simply a list of context names in the order in which

they will be searched when symbols are encountered.[1] The current context is always searched first. Continuing the analogy with file system directories, the context search path is analogous to the PATH environment variable of UNIX and DOS systems, while the current context is analogous to the working directory.

The context search path is kept in the system variable `$ContextPath`.

```
$ContextPath
{Global`, System`}
```

This shows that contexts will be searched in the following order: the current context, the `Global`` context, and finally the `System`` context.

For example, when the symbol `I` is used, and the current context is `Global``, the system does not find a symbol called `Global`I` so it uses the symbol `System`I`.

```
?I
I represents the imaginary unit Sqrt[-1].
```

If we were to define a symbol in the `Global`` context named `I`, then we would no longer be able to access `System`I` by its short name.

```
Global`I = 2
I::shdw: Symbol I appears in multiple contexts
    {Global`, System`}; definitions in context Global`
     may shadow or be shadowed by other definitions.
2
```

The warning message says that this definition has shadowed another symbol having the same name, that is, it has made that symbol inaccessible by its short name. (We'll discuss shadowing in detail in Section 8.4.)

There are now two distinct symbols named `I`.

```
?*`I
I          System`I
```

Because the current context is `Global``, an unqualified `I` refers to `Global`I`.

```
I^2
4
```

You can always refer to the "real" `I` (the imaginary unit) by using its long name.

```
System`I^2
-1
```

If you were to set the current context to `System`` ...

```
Begin["System`"]
System`
```

then `I` refers to `System`I`.

```
I^2
-1
```

1. Do not confuse the context search path `$ContextPath` with the directory search path `$Path`, which contains the list of directories that the kernel searches when attempting to locate a file.

This leaves the System` context.

```
End[]
System`
```

We really should get rid of the bogus definition of I. However, Clear will not do it, as it removes only the *values* of a symbol, and not the symbol itself.

```
Clear[I]
Context[I]
Global`
```

The correct function for this job is Remove:

```
Remove[I]
Context[I]
System`
```

8.2 Package Mechanics

Now that you understand contexts, you have almost all of the tools necessary to write a package. There are two other context-related functions, BeginPackage and EndPackage, that make context management for a package much easier. After discussing these, we will develop a complete example package.

8.2.1 BeginPackage and EndPackage

The first thing that every package must do is to call BeginPackage, which creates a new context and changes the context search path to consist of only the newly defined context and the System` context:

The current context is changed to example`.

```
BeginPackage["example`"]
example`
```

And the context search path is changed as shown.

```
$ContextPath
{example`, System`}
```

BeginPackage temporarily removes all other contexts except System` from the context search path, which guarantees that symbols defined by the package will be distinct from symbols defined by other packages or by the user.[2] The System` context is left on the context search path so that the package can use built-in functions.[3]

Here are some symbols created in the package context:

2. This assumes, of course, that the user does not load two different packages that define the same context. Thus, the problem has not been "solved," but merely pushed to a higher level. In practice, however, this is all that's necessary.
3. If a package needs the services of other packages, a mechanism exists for allowing this; see Section 8.2.3.

```
f = "a package symbol";
g = "another package symbol";
```

It's considered good programming practice to place all definitions that are not meant to be seen by the user into a private subcontext of the package context. By convention, this context is called *packagecontext*`Private`.

The context mark preceding this context name indicates that it is nested within the current context.

```
Begin["`Private`"]
example`Private`

a = "a very shy symbol";
```

Exit from the private subcontext.

```
End[]
example`Private`
```

Because the subcontext was entered using Begin, it was not added to the context search path. Thus, a user will not *inadvertently* see or alter any of the data structures declared inside of this subcontext. (Of course, a user could deliberately subvert this protection by referring explicitly to example`Private`a, but that's not the programmer's concern.)

EndPackage restores the value of $Context to its previous value and sets $ContextPath to its previous value with the name of the new context prepended to it.

```
EndPackage[]
example`

$Context
Global`

$ContextPath
{example`, Global`, System`}
```

Prepending the new context name to the context path means that symbols defined in the package context can be referred to by their short names:

```
?f
example`f
f = "a package symbol"
```

Since the private subcontext is not on the context search path, symbols in that context are "hidden" from the user.

```
?a
Information::notfound: Symbol a not found.
```

Note that since the private subcontext is a subcontext of the package context, it is distinct from the private subcontext of any other package that follows the canonical package structure outlined here.

8.2.2 An example package

The example package developed in this section generates pseudorandom numbers using the linear congruential generator $x_{n+1} = 7^5 x_n \bmod(2^{31} - 1)$ ([Jain 91] example 26.3).

The first thing that the package needs to do is to call `BeginPackage`:

```
BeginPackage["LCGRandom`"]
LCGRandom`
```

The next thing to do is to create any symbols that will be *exported*, i.e., visible outside of the package. This is conventionally done by defining *usage messages* for these symbols. These are the messages that will be printed when the user evaluates `?symbolname`.

The package exports two functions.

```
LCGSetSeed::usage = "LCGSetSeed[x] sets the LCG\
 random number generator's seed to the integer x.";
LCGRandom::usage = "LCGRandom[] generates a\
 uniformly distributed random number in the range\
 {0.,1.} using a linear congruential generator.";
```

Here's an example of a usage message at work. (This code is not part of the package.)

```
?LCGRandom
LCGRandom[] generates a uniformly distributed random
   number in the range {0.,1.} using a linear
   congruential generator.
```

The implementation of the exported functions is done inside of a private subcontext.

```
Begin["`Private`"]
LCGRandom`Private`
```

Private data are defined first. To speed up the generator, we precompute the constant $2^{31} - 1$ and store it in a symbol called `modulus`. The symbol `seed` is used to keep track of the most recent value returned by the generator.

```
modulus = 2^31 - 1;
seed = 1;
```

This is also the place to define any error messages for the functions defined in the package. We will defer a discussion of error messages to Section 9.1, "Diagnostic Messages."

Now for the actual implementation of the two functions exported by the package:

The seed must lie between 1 and `modulus - 1`.

```
LCGSetSeed[x_Integer /; 1 <= seed < modulus] :=
    seed = x
```

`LCGRandom` takes no parameters; it uses the package's internal state.

```
LCGRandom[] := (
    seed = Mod[7^5*seed, modulus];
    N[seed/modulus]
)
```

Now that the implementation is finished, leave the `Private` context.

```
End[]
LCGRandom`Private`
```

We are now back in the package context. (This code is not part of the package.)

```
$Context
LCGRandom`
```

Next, protect the functions that have been defined.

```
Protect[LCGSetSeed, LCGRandom]
{LCGSetSeed, LCGRandom}
```

Finally, end the package with `EndPackage`.

```
EndPackage[];
```

`EndPackage` restores the current context and the context search path to whatever they were before the call to `BeginPackage`. However, the context `LCGRandom`` is left at the front of the context path. This means that users can refer to the symbols that the package exports by their short names.

```
{$Context, $ContextPath}
{Global`, {LCGRandom`, example`, Global`, System`}}

?LCGRandom`*
LCGRandom  LCGSetSeed
```

Let's test the package:

This generates 1000 pseudorandom numbers between 0 and 1.

```
LCGSetSeed[4446290];
test = Table[LCGRandom[], {1000}];
Short[test]
{0.798307, 0.152043, 0.388023, <<996>>, 0.455062}
```

The mean of these 1000 numbers is close to 1/2.

```
Plus @@ test / Length[test]
0.506407
```

Everything works just as a built-in function.

```
?LCGRandom
LCGRandom[] generates a uniformly distributed random
   number in the range {0.,1.} using a linear
   congruential generator.
```

Users cannot modify the definitions of `LCGRandom` and `LCGSetSeed`.

```
Clear[LCGRandom]
Clear::wrsym: Symbol LCGRandom is Protected.
```

The symbols in the `LCGRandom`Private`` context are not accessible.

```
?modulus
Information::notfound: Symbol modulus not found.
```

The full definition of the package's functions can be viewed using the `??` operator. If this is not desired, give these functions the attribute `ReadProtected`.

Note that the private symbols print as long names, since the context `LCGRandom`Private`` is not on the context search path.

```
??LCGSetSeed
LCGSetSeed[x] sets the LCG random number generator's
   seed to the integer x.
Attributes[LCGSetSeed] = {Protected}
LCGSetSeed[LCGRandom`Private`x_Integer /;
    Inequality[1, LessEqual, LCGRandom`Private`seed,
     Less, LCGRandom`Private`modulus]] :=
  LCGRandom`Private`seed = LCGRandom`Private`x
```

The last thing to do, of course, is to put these definitions into a file called *LCGRandom.m* and place this file in a directory that *Mathematica* will search when it looks for packages.[4] You don't have to put the file in the standard *Packages* directory, but if you don't, you have to tell *Mathematica* where to look for it. The search path for files is kept in the system variable `$Path`. You can modify this variable to make *Mathematica* look in other directories.

(These are Macintosh pathnames.)

```
$Path
{., Sartorius:Mathematica 3.0,
  Sartorius:Mathematica 3.0:Packages:Applications,
  Sartorius:Mathematica 3.0:Packages:Standard,
  Sartorius:Mathematica 3.0:Configuration:Kernel,
  Sartorius:Mathematica\
   3.0:SystemFiles:Graphics:TextResources}
```

The package can then be loaded using the command `Needs["LCGRandom`"]` or `Get["LCGRandom`"]`. The advantage of using `Needs` rather than `Get` is that `Needs` checks the system variable `$Packages` to ensure that the package has not already been loaded, and then adds the package name to `$Packages`. This is important since most packages are not designed to be loaded more than once in the same *Mathematica* session; doing so usually causes a barrage of error messages.

Both `Needs` and `Get` translate context names to filenames using the function `ContextToFilename`. The translation is done in a way that is suitable for the particular operating system that is being used. For example, `ContextToFilename["aaa`bbb`"]` returns the string `"aaa/bbb.m"` on a UNIX system; `"aaa\bbb.m"` on a DOS-based system; and `"aaa:bbb.m"` on a MacOS system. If you create a package whose context name is more than eight characters long for use on a DOS-based system, you may wish to add a new rule for `ContextToFilename` to do the translation correctly, e.g.,

```
Unprotect[ContextToFilename];
ContextToFilename["verylongname`"] := "verylong.m"
```

4. In version 3.0, you also can use the `DumpSave` command to save the definition of a package in a binary format. See Section 12.2.3, "Save and DumpSave," for details.

This definition should be placed in the package user's *init.m* file so that it is available in every session. Alternatively, the user can load the package using this syntax:

The second argument tells Needs the name of the file in which the requested context is defined.

```
Needs["verylongname`", "verylong.m"]
```

Exercises

1. Create a package file for `LCGRandom`, set `$Path` appropriately, and verify that you can load the package using `Needs`. Note that if you have been following the development of the package in your own *Mathematica* session, you will need to restart the kernel to wipe out the existing definitions of the `LCGRandom`` context.
2. Write a package that extends the functionality of the built-in `Dot` function so that it always multiplies chains of matrices in the optimal order (see Section 6.4.2, "Application: Matrix-chain multiplication"). A bonus of wrapping this algorithm inside a package is that it prevents the `Global`` context from being polluted by all of the cached results created by the algorithm.

8.2.3 Using other packages

If a package depends on some other package, there are two ways to ensure that the other package is available at the time the package is loaded: *normal import* and *hidden import*.

To import packages in the normal way, add the context names of those packages to the `BeginPackage` call. For example, the following `BeginPackage` statement loads the context `NeededPackage`` (if necessary) and places it on the context path right after `MyPackage``:

```
BeginPackage["MyPackage`", "NeededPackage`"]
```

Now `MyPackage`` can use any symbols defined in `NeededPackage``.

One possible drawback to this method is that it "pollutes" the context path with extra contexts. The user may stumble across a symbol that is defined in a subsidiary package, which she or he isn't even aware has been loaded. This has the potential to cause some confusion.

To avoid this problem, the subsidiary contexts can be loaded, using `Needs`, *after* the call to `BeginPackage`; this is called hidden import [Maeder 91]. Now the package can still use all of the symbols defined in the other packages, but after the `EndPackage` statement is executed, the other packages will not appear on the context path (but they will appear in `$Packages`).

8.3 Stylistic Considerations

There are quite a few stylistic considerations to which you should pay heed when you write a package.

8.3.1 Naming conventions

The general naming convention for symbols used in *Mathematica* is that names should be capitalized English words or concatenations of such words. Names should be descriptive and fully spelled out. Abbreviations should not be used unless they are in widespread use (e.g., `Det` instead of `Determinant`).

Function names should be unique whenever possible; however, if you define a function whose purpose is similar to a built-in function, its name should be similar to that of the built-in function. Avoid adding definitions to built-in functions unless absolutely necessary, as this may lead to unexpected results in some cases and may slow down the evaluation of a wide class of expressions. Also, try to avoid defining two different functions when a single function with an option would do.

Package filenames should be unique within the first eight characters in order to avoid problems on DOS-based file systems.

8.3.2 Export a minimal interface

As discussed earlier in this chapter, a package should encapsulate the details of its implementation inside of a private subcontext. Moreover, even the *names* of functions and symbols that are not of immediate utility to the user should be hidden; this avoids confusion on the user's part. A good rule to follow is that only those symbols that have usage messages should be visible outside of the package. Any auxiliary functions and/or temporary variables (such as `modulus` and `seed` in the `LCGRandom`` example) should be created inside of a private subcontext.

The package *EscapingSymbols.m* (*MathSource* item #0204-961-0022) can help you to identify symbols defined by a package that "escape" from it without usage messages.

```
Needs["EscapingSymbols`",
"MathSource:Enhancements:Language:0204-961;
Developer Tools for A:EscapingSymbols.m"]

?EscapingSymbols

EscapingSymbols[expr] returns a list of symbols
   escaping from the execution of expr (i.e., new
   symbols visible to the user but lacking usage
   messages).
```

The file `example.m` contains a simple package that we will use to demonstrate the action of `EscapingSymbols`.

The syntax `!!file` displays the contents of `file` without evaluating them.

```
!!example.m
BeginPackage["example`"];
    good::usage = "I'm a good symbol";
    bad = "I'm a bad symbol";
    Begin["`Private`"];
        irrelevant = "I don't matter";
    End[];
EndPackage[];
```

Now the package is loaded.

```
EscapingSymbols[<<example.m]
{bad}
```

The symbol `bad` is flagged since it is visible but has no usage message. The symbol `irrelevant` is exactly that, because it is in a private context that isn't visible to the user.

8.3.3 Usage message conventions

Every usage message should begin with a syntax declaration. This is because some versions of the front end have a command called **Make Template** that pastes generic arguments for a function into the notebook following the name of the function (see your *User's Guide* for details). This is very handy for functions that take several arguments, the order of which is uncertain to the user. The template mechanism looks at only the *beginning* of a usage message for a template, hence the first thing in the usage message should be a usable template. Compare the following two examples:

This is a good usage message.

```
LCGSetSeed::usage = "LCGSetSeed[x] sets the LCG
    random number generator's seed to the integer x.";
```

This is a bad usage message.

```
LCGSetSeed::usage = "LCGSetSeed is used to set the
    seed for the LCG random number generator.
    LCGSetSeed[x] sets the seed to x.";
```

8.3.4 Diagnostic messages

Never use `Print` to issue diagnostics; use `Message` instead. There are several good reasons for this, which we'll discuss in detail in Section 9.1, "Diagnostic Messages."

8.3.5 Package documentation

If you look inside any of the standard packages, you will see many comments at the beginning of the file that have the following general format:

```
(* :Name: Calculus`Limit` *)

(* :Author: Victor S. Adamchik, Summer 1991 *)

(* :Summary:
```

```
        This package provides an enhancement to the
        built-in Limit. It allows one to find the limits of
        expressions containing elements of a wide class of
        elementary and special functions.
*)
```

Comments with this special structure are called *annotations*. The words or phrases surrounded by colons are called *keywords*. Typical keywords that you might find in a standard package are `Name` (or `Context`), `Author`, `Version`, `History`, `Limitations`, and `Warnings`. Standard keywords are described in [WRI 93d].

Keywords and annotations can be extracted from a package using the `Annotation` function defined in the standard package `Utilities`Package`. All packages should include annotations so that users can get information about them by using these utilities. For example:

```
Needs["Utilities`Package`"]
```

This function searches a specified path (or list of paths) for packages matching a search string.

```
FindPackages[$Path, "*Limit*"]
{{}, {}, {}, {Calculus`Limit`,
   NumericalMath`NLimit`}, {}, {}}
```

This finds all of the annotative keywords in the package.

```
Annotation["Calculus`Limit`"]
{Name, Author, Mathematica Version, Context,
  Keywords, Copyright, Summary, Requirements,
  Limitations}

Annotation["Calculus`Limit`", "Author"] // InputForm
{"(* :Author: Victor S. Adamchik, Summer 1991 *)"}
```

Like the other standard packages, `Utilities`Package`` is documented in [WRI 93b].

8.4 Advanced Topic: Shadowing

Shadowing occurs when more than one context on the context search path contains a symbol of a given name. The most common cause of shadowing, which will be illustrated in Section 8.4.1, is attempting to use a symbol from a package before loading that package.

Several techniques and tools exist for preventing or at least minimizing the shadowing problem. Some of these are aimed at package developers (Sections 8.4.2 and 8.4.3). In Section 8.4.4 we demonstrate a package called *Unshadow.m*, available from *MathSource*, that provides a tool for removing symbols that shadow other symbols. Finally, in Section 8.4.5 we develop a new package that avoids shadowing *automatically*.

Digression: CleanSlate

Since we will be doing a lot of experimentation with context creation, it will be convenient to have a mechanism for removing entire contexts from a kernel session without having to quit and restart the kernel. The `CleanSlate` package [Gayley 93a] provides this functionality. If you intend to work through the examples in the remainder of this chapter, you are advised to load this package. Otherwise, you will find it necessary to quit and restart the kernel several times in order to wipe out certain contexts.

The `CleanSlate` function is able to remove only those symbols and contexts that are created after its package is loaded. Therefore, we terminate the current kernel session at this point; the following command is the first command in a new kernel session.

```
Needs["CleanSlate`",
"MathSource:Enhancements:System:0204-310;
CleanSlate Package:CleanSlate.m"]

?CleanSlate
CleanSlate[] purges all symbols and their values in
   all contexts that have been added to the context
   search path ($ContextPath), since the CleanSlate
   package was read in. This includes user-defined
   symbols (in the Global` context) as well as any
   packages that may have been read in. It also
   removes most, but possibly not all, of the
   additional rules for System symbols that these
   packages may have defined. It also clears the In[]
   and Out[] values, and resets the $Line number, so
   new input begins as In[1].
   CleanSlate["Context1`","Context2`"] purges only the
   listed contexts.
```

This option suppresses certain status information.

```
SetOptions[CleanSlate, Verbose->False]
```

The author highly recommends the `CleanSlate` package. Remember, `CleanSlate` can remove only those symbols that are created after `CleanSlate` is loaded. Therefore, it is best to insert a command to load the package in your *init.m* file, where it will be executed automatically each time the kernel starts up.

8.4.1 The shadowing problem

Every *Mathematica* user has, at one time or another, made a mistake like the following:

A package symbol is referred to before the package is loaded.

```
Show[Polyhedron[Dodecahedron]]
Show::gtype: Polyhedron is not a type of graphics.
Show[Polyhedron[Dodecahedron]]
```

The package is loaded and strange warning messages appear.

```
Needs["Graphics`Polyhedra`"]
Polyhedron::shdw:
   Symbol Polyhedron appears in multiple contexts
    {Graphics`Polyhedra`, Global`}; definitions in
     context Graphics`Polyhedra`
     may shadow or be shadowed by other definitions.
Dodecahedron::shdw:
   Symbol Dodecahedron appears in multiple contexts
    {Graphics`Polyhedra`, Global`}; definitions in
     context Graphics`Polyhedra`
     may shadow or be shadowed by other definitions.
```

It appears that *Mathematica* still doesn't know about these symbols.

```
Show[Polyhedron[Dodecahedron]]
Show::gtype: Polyhedron is not a type of graphics.
Show[Polyhedron[Dodecahedron]]
```

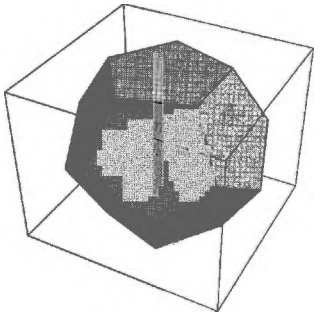
This problem, which is called shadowing, results from the fact that the initial use of the symbol `Polyhedron` (and `Dodecahedron`) created a symbol of that name in the `Global`` context.

```
?*`Polyhedron
Polyhedron
Graphics`Polyhedra`Polyhedron
```

Since the `Global`` context is the current context, it is searched before `Graphics`Polyhedra``, so the package's definition of `Polyhedron` is not found.

Experienced users know that to remedy this situation, all that is necessary is to eliminate the offending symbols by using the `Remove` function:

```
Remove[Polyhedron, Dodecahedron]

Show[Polyhedron[Dodecahedron]]
```



```
-Graphics3D-
```

Shadowing can be a major annoyance if very many symbols are involved. Furthermore, judging from the frequency with which questions about this topic appear in the

Mathematica Internet discussion group (`comp.soft-sys.math.mathematica`), shadowing is a common and frustrating problem for inexperienced *Mathematica* users.

"Clean house" before continuing.

```
CleanSlate[];
```

8.4.2 Designing packages to prevent shadowing

Package designers can spare users of their packages the frustration of shadowing by using one of the following tricks. Unfortunately, each of these techniques has drawbacks as well.

Roman Maeder [Maeder 91] suggests including the `Global`` context in the `BeginPackage` command:

```
BeginPackage["MyPackage`", "Global`" ..]
```

This adds the `Global`` context to the context search path at the time the package is loaded. Any symbol created by the package that does not already exist in the `Global`` context will be created in the `MyPackage`` context. If such a symbol already exists in the `Global`` context (because a user tried to use it too soon), then that symbol — in the `Global`` context — will be redefined by the package. Hence, there will be only one symbol with the given name, and no shadowing will occur.

Of course, Maeder's trick is a calculated risk, since the user may have deliberately associated a definition with one of the conflicting symbol names, and the package might wipe out that definition. To get around this problem, Nancy Blachman [Blachman 92] recommends starting a package with the following code:

```
BeginPackage["MyPackage`"]
EndPackage[]
```

The purpose of this device is to prepend `MyPackage`` to the context search path. Now continue with the rest of the package; all symbols will be created in the context in which `Needs` was called, which *probably* is `Global`. (Be sure to declare the package's private subcontext as `MyPackage`Private`` rather than just `` `Private` ``, to ensure that the package's private symbols do not end up in a context called `Global`Private`.)

It may seem as though this method has the same potential for problems as Maeder's. The difference is that if a sophisticated user wishes to create the package's symbols in a different context — which cannot be done using Maeder's method — he or she can always do this:

```
BeginPackage["NewContext`"]
Needs["MyPackage`"]
EndPackage[]
```

This prepends `NewContext`` to the context search path and causes all nonprivate symbols defined by `MyPackage` to be created in that context. Thus, the user's definitions of any conflicting symbols will be preserved (although they will still shadow the definitions from the package.)

Exercise

1. Rewrite the `LCGRandom`` package using each of the techniques of this section. Execute the commands in the package one at a time, inspecting the values of `$Context` and `$ContextPath` each time a context-related command is executed. Note that you will have to start a fresh kernel before each reload of the package (or use the `CleanSlate` function).

8.4.3 Predeclaring package symbols

Every standard package directory (e.g., *Graphics*) contains a file called *Master.m*, which contains declarations for each package contained in that directory (e.g., `Graphics`Graphics``, `Graphics`Graphics3D``, etc.). These declarations have the following general form:

```
DeclarePackage["Graphics`Polyhedra`",
    {"Polyhedron", "Dodecahedron"}];
```

This declaration doesn't actually load the `Graphics`Polyhedra`` package; it simply tells the kernel that the symbols `Polyhedron` and `Dodecahedron` can be found there. The symbols are created in the package context and given the `Stub` attribute:

```
?Polyhedron
Graphics`Polyhedra`Polyhedron
Attributes[Polyhedron] = {Stub}
Polyhedron = ""
```

The `Stub` attribute signifies that the package that defines a symbol has not yet been loaded. Upon the first use of any symbol having this attribute, the kernel will automatically load the package that defines that symbol.

Using the symbol `Polyhedron` causes the `Graphics`Polyhedra`` package to load.

```
Polyhedron
Polyhedron

?Polyhedron
Polyhedron[name] gives a Graphics3D object
   representing the specified solid centered at the
   origin and with unit distance to the midpoints of
   the edges.  Polyhedron[name, center, size] uses the
   given center and size. The possible names are in
   the list Polyhedra.
```

If there are certain standard packages that you use on occasion but which would occupy too much memory if they were all loaded simultaneously, you can avoid any

possibility of shadowing problems with those packages by loading their master packages at the beginning of each *Mathematica* session. You can automate this procedure by putting the statements to load the master packages into your *init.m* file; see the documentation for your version of *Mathematica* for the location of this file.

If you are a *Mathematica* developer and you are writing a particularly large package that uses a lot of memory, or a set of interrelated packages, you may want to create a master package as a convenience for your users. A master package should be structured as follows, with one `DeclarePackage` statement for each component package:

```
BeginPackage["MyPackage`Master`"];
EndPackage[];
DeclarePackage["Package1", {symbol1, symbol2, ...}];
DeclarePackage["Package2", {symbol3, symbol4, ...}];
```

This and other techniques for organizing a large set of interrelated packages are exemplified by the standard `Statistics`` packages.

Of course, the master package can prevent shadowing of package symbols only if it has been loaded beforehand! Users who have shadowing problems because they forget to load a standard package often forget to load the master package as well.

Exercise

1. Write and test a master package for the `LCGRandom`` package.

8.4.4 Removing shadows after the fact

Shadowing happens would make a perfect bumper sticker for a *Mathematica* user. The fact is, the vast majority of packages — standard packages included — do not take any of the steps described in Section 8.4.2 for avoiding the shadowing problem. Each of the standard package directories does provide a master package; however, it's often the case that users forget to load them.

In response to this situation, Ulrich Jentschura has written a package called *Unshadow.m* (*MathSource* item #0204-815), which defines a function called `Unshadow` that seeks out and destroys *all* global symbols that are shadowing other symbols. This is especially useful in cases where the user has shadowed so many symbols that the error message mechanism suppresses some of the *sym*`::shdw` messages! Here is a demonstration of using `Unshadow` to rectify the situation illustrated in Section 8.4.1.

First, remove the `Graphics`Polyhedra`` context from the current session.

```
CleanSlate["Graphics`Polyhedra`"];
```

Now make the same mistake as before.

```
g = Polyhedron[Dodecahedron];
Show[g]
Show::gtype: Polyhedron is not a type of graphics.
Show[Polyhedron[Dodecahedron]]
```

The full text of these messages has been edited out for brevity.

```
Needs["Graphics`Polyhedra`"]
Polyhedron::shdw: ...
Dodecahedron::shdw: ...
```

Load the `Unshadow` package.

```
Needs["Unshadow`", "MathSource:Enhancements:System:\
0204-815; Unshadow Package:Unshadow.m"]
Unshadow Package, Version of April, 1993
For information, type ?Unshadow
```

`Unshadow` returns a list of the symbols that have been removed.

```
Unshadow[]
The following symbols appeared both in the global and
  in another context.
They have been removed in the global context
  to prevent shadowing.
{Dodecahedron, Polyhedron}
```

Indeed, the shadowing symbols are gone.

```
Context /@ {Polyhedron, Dodecahedron}
{Graphics`Polyhedra`, Graphics`Polyhedra`}
```

8.4.5 "Smart" shadow removal

`Unshadow` does not make any attempt to distinguish between symbols that the user might or might not need. One obvious way to do this would be simply to return the list of shadowing symbols without removing them; the user could then verify that none of the symbols are important and execute the command `Remove @@ %`. Alternatively, `Unshadow` could prompt the user for confirmation (using the `Input` function) before removing each symbol. Better yet, it could give the user the best of both worlds by providing an option that controls its behavior in this regard.

On the other hand, there is something to be said for making the process as automatic as possible. A more sophisticated approach that subscribes to this philosophy would be to determine whether or not the shadowing symbol has any values; if it does not, then it probably is not important and can be removed automatically. This is the approach that we shall take next. Our efforts will culminate in a new package, `AntiShadow`, that removes shadows during the package-loading process, taking care never to remove a symbol that has any values.

Determining if a symbol has values

A symbol can have many different kinds of values. For example, this assignment creates an ownvalue:

```
test := 1/0
```

`OwnValues[sym]` lists the ownvalue of a symbol.

```
OwnValues[test]
{HoldPattern[test] :> 1/0}
```

There are five other kinds of rules (described in Table 7-1, "The Six Types of Rules for a Symbol," on page 186) that can be attached to a symbol: downvalues, upvalues, subvalues, format values, and Nvalues. In addition to these, we will check for the existence of attributes, options, and messages.

If a symbol has the `ReadProtected` attribute, `OwnValues` and friends will cause an error message to be printed. Therefore, we must check for attributes before checking any of the other types of values. Furthermore, `Options` does not hold its argument; therefore, we can check for options only after determining that the symbol has no rule values. Here is a function that does all of this:

```
SetAttributes[hasvalues, HoldFirst];
hasvalues[s_Symbol] :=
    Attributes[s] =!= {} ||
    OwnValues[s] =!= {} ||
    UpValues[s] =!= {} ||
    DownValues[s] =!= {} ||
    SubValues[s] =!= {} ||
    NValues[s] =!= {} ||
    FormatValues[s] =!= {} ||
    Options[s] =!= {} ||
    Messages[s] =!= {}
```

The `hasvalues` function is given the `HoldFirst` attribute so that its argument will not evaluate. Here is an example:

```
hasvalues[test]
True
```

The `Unshadow` package could easily be enhanced to remove only those symbols for which `hasvalues` returns `False`. Experimentation with the `Unshadow` package is left to the exercises. Rather than modify that package, we will develop an entirely new package called `AntiShadow`.

Like the `Unshadow` package, the `AntiShadow` package will be working with symbol names in string form, for reasons that will become clear later. Unfortunately, `OwnValues` and the like do not accept strings as arguments.

```
OwnValues["test"]
OwnValues::sym:
   Argument test at position 1
     is expected to be a symbol.
OwnValues[test]
```

We could convert the string to an expression using `ToExpression`, but that would result in an evaluation of the symbol.

```
ToExpression["test"]

                                        1
Power::infy: Infinite expression  -  encountered.
                                        0

ComplexInfinity
```

The solution to this problem is to use `ToHeldExpression` to convert the string to an expression and wrap `Hold` around it at the same time:

```
ToHeldExpression["test"]
Hold[test]
```

The ownvalues of the symbol can then be found by using `Apply` to replace the head `Hold` with the head `OwnValues`.

```
OwnValues @@ %

{HoldPattern[test] :> 1/0}
```

The `hasvalues` function can be made to work on symbol names by using the exact same technique.

```
hasvalues @@ ToHeldExpression["test"]
True
```

There still remains another, more subtle problem. Even if a symbol has no values, removing it can have adverse consequences on other symbols, such as the symbol `g` defined in the last section:

g still is a function of the removed symbols.

```
g
Removed[Polyhedron][Removed[Dodecahedron]]
```

The expressions of the form `Removed[`*name*`]` aren't normal expressions; in fact, they are a pathological output format for symbols that have been marked internally for removal:

```
FullForm[ g[[1]] ]
Removed["Dodecahedron"]

Head[%]
Symbol
```

This means that the pattern `Removed[x_]` will not match the removed symbols.

```
g /. Removed[x_] :> ToExpression[x]
Removed[Polyhedron][Removed[Dodecahedron]]
```

But we can extract the removed symbols from `g` using the pattern `_Symbol`, and then we can use the extracted symbols as the left-hand sides of replacement rules:

```
Cases[g, _Symbol, Infinity, Heads -> True]
{Removed[Polyhedron], Removed[Dodecahedron]}

Thread[Rule[%, {Polyhedron, Dodecahedron}]]
{Removed[Polyhedron] -> Polyhedron,
  Removed[Dodecahedron] -> Dodecahedron}

g /. %
-Graphics3D-
```

It is even possible to prevent the removal of symbols upon which other symbols depend by using the `dependson` function developed in Section 7.3.7. `dependson` could be applied to every symbol in the `Global`` context to create a list of all symbols upon which any global symbol depends, and the resulting symbols could be eliminated from the list of candidates for removal. The drawback of this approach is that, in a *Mathematica* session with a lot of symbols, it could slow down the shadow removal process quite dramatically. We will not pursue this idea any further here; it is left as an exercise for the interested reader.

The next call to `CleanSlate` removes only the `Graphics`Polyhedra`` context (because we still need the definition of `hasvalues`).

```
CleanSlate["Graphics`Polyhedra`"];
```

Avoiding shadowing dynamically

To make the process of eliminating shadowing symbols as automatic as possible, we would like to remove the offending symbols without requiring *any* action on the part of the user. The observation that makes this possible is that, whenever a symbol is defined that might shadow some other symbol, the message `General::shdw` is generated:

```
General::shdw
Symbol `1` appears in multiple contexts `2`;\
  definitions in context `3` may shadow or be\
  shadowed by other definitions.
```

Therefore, our next course of action will be to intercept calls to `Message[sym_::shdw, ___]`.

From the text of the `General::shdw` message we deduce that `Message` will be called with four arguments: the name of the message plus the three string substitution arguments. The following rule for `Message` allows us to see the full form of each of these arguments:

```
Unprotect[Message];
Message[arg0_, arg1_, arg2_, arg3_] := (
    Print[FullForm[#]]& /@ {arg0, arg1, arg2, arg3};
    Print[""]
)
```

Next, we recreate the problematic global symbols and load the package.

```
Show[Polyhedron[Dodecahedron]];
Show::gtype: Polyhedron is not a type of graphics.

Needs["Graphics`Polyhedra`"]
MessageName[Polyhedron, "shdw"]
HoldForm["Polyhedron"]
HoldForm[List["Graphics`Polyhedra`", "Global`"]]
HoldForm["Graphics`Polyhedra`"]

MessageName[Dodecahedron, "shdw"]
HoldForm["Dodecahedron"]
HoldForm[List["Graphics`Polyhedra`", "Global`"]]
HoldForm["Graphics`Polyhedra`"]
```

The first argument is a `MessageName` object. Since `MessageName` has the `HoldFirst` attribute, the symbol inside of it does not evaluate. The rest of the arguments, which are substituted for `` `1` ``, `` `2` ``, etc., are strings wrapped in `HoldForm`. Let's refine our definition for `Message` to use destructuring to extract the symbol, symbol name, and context names. We will, of course, keep the symbol itself wrapped in `HoldForm` to prevent its evaluation.

```
Clear[Message]
Message[sym_::shdw, HoldForm[symName_],
        HoldForm[cxNames_], HoldForm[newcxName_]] :=
(   Print["sym=", HoldForm[sym]];
    Print["symName=", InputForm[symName]];
    Print["cxNames=", InputForm[cxNames]];
    Print["newcxName=", InputForm[newcxName], "\n"];
)

CleanSlate["Graphics`Polyhedra`"];
Needs["Graphics`Polyhedra`"]
sym=Polyhedron
symName="Polyhedron"
cxNames={"Graphics`Polyhedra`", "Global`"}
newcxName="Graphics`Polyhedra`"

sym=Dodecahedron
symName="Dodecahedron"
cxNames={"Graphics`Polyhedra`", "Global`"}
newcxName="Graphics`Polyhedra`"
```

One might think that we could use `hasvalues[sym]` to check if the shadowing symbol has any definitions in the `Global`` context. However, this will not work, because at the time that `hasvalues` is called, the current context will be the new package's context, not `Global`. Instead, we must construct the long name `"Global`" <> symName` explicitly and use `hasvalues @@ ToHeldExpression[#]&` on it.

We also can use the symName argument to generate our own warning messages to the user whenever we remove a symbol to avoid a shadow. Here is the text of the message that will be generated:

```
General::noshdw =
    "Warning: the symbol `1` has been removed from
     the global context to prevent shadowing.";
```

Here, then, is the definition of Message that will do the trick.

```
Clear[Message]
Message[sym_::shdw, HoldForm[symName_],
        HoldForm[cxNames_], HoldForm[newcxName_]] /;
    MemberQ[cxNames, "Global`"] &&
    !hasvalues@@ToHeldExpression["Global`"<>symName] :=
With[{globalsym = "Global`"<>symName},
    Message[sym::noshdw, globalsym];
    Remove[globalsym]
]
```

If Global` is one of the contexts containing the symbol named symName (MemberQ[cxNames, "Global`"]), and Global`symName has no values (!hasvalues @@ ToHeldExpression["Global`" <> symName]), the warning message is printed and Global`symName is removed.

Here is a demonstration. We create the symbols Polyhedron and Dodecahedron in the Global` context, and we give the latter a value.

```
Polyhedron[Dodecahedron];
Dodecahedron = "I have a value!";
```

Our rule intercepts the call to Message[Polyhedron::shdw].

```
CleanSlate["Graphics`Polyhedra`"];
Needs["Graphics`Polyhedra`"]
Polyhedron::noshdw:
   Warning: the symbol Global`Polyhedron
     has been removed from the global context to
     prevent shadowing.
Dodecahedron::shdw:
   Warning: Symbol Dodecahedron
     appears in multiple contexts
    {Graphics`Polyhedra`, Global`}; definitions in
     context Graphics`Polyhedra`
     may shadow or be shadowed by other definitions.
```

Global`Polyhedron has been removed automatically.

```
?*`Polyhedron
Polyhedron[name] gives a Graphics3D object
   representing the specified solid centered at the
   origin and with unit distance to the midpoints of
   the edges.  Polyhedron[name, center, size] uses the
   given center and size. The possible names are in
   the list Polyhedra.
```

On the other hand, `Global`Dodecahedron` has not been removed, since the existence of values for it implies that removing it might not be what the user wants. In the given scenario, it's up to the user to decide whether or not to remove `Global`Dodecahedron`.

```
?*`Dodecahedron

Dodecahedron
Graphics`Polyhedra`Dodecahedron
```

Now that we're satisfied with the new behavior of `Message`, we should reprotect it.

```
Protect[Message];
```

You will use the definitions developed in this section in one of the exercises to construct the `AntiShadow` package. Since the new rule for `Message` can perform its function only if it already exists at the time that a package is loaded, `AntiShadow` will be most effective if it is loaded automatically at the beginning of every *Mathematica* session by *init.m*.

Exercises

1. Create a package to encapsulate the definitions developed in this section. Note that the package doesn't export any new symbols. Nevertheless, you should use `BeginPackage` and `EndPackage` so that the package name appears in `$Packages`.
2. There's a subtle point about the new rule for `Message`: If the user ever deliberately creates a symbol in the `Global`` context that would shadow a symbol in some other context, she or he won't be allowed to!

```
Global`GreatStellatedDodecahedron

GreatStellatedDodecahedron::noshdw:
   Warning: the symbol
    Global`GreatStellatedDodecahedron
     has been removed from the global context to
     prevent shadowing.

Removed[GreatStellatedDodecahedron]
```

   The return value is a reference to the symbol just created and subsequently removed. This is either a bug or a feature, depending on one's point of view. If you take the former view, it is quite easy to fix: Simply add another condition to the new rule for `Message` that checks to see if `newcxName` is not equal to `"Global`"`.

   More generally, delete `newcxName` from the list of contexts `cxNames` in which the symbol appears; if the result still contains the string `"Global`"`, then it's okay to go ahead and remove `Global`sym`. Implement and test this bug-fix.
3. Obtain the package *Unshadow.m* and implement some of the suggested improvements.

4. (Nontrivial project!) To be even more cautious about which symbols are removed, you could check to see that not only does a symbol have no values, but that in addition *no other symbol depends upon this symbol.*

   The rule for `Message` will need to check the values of *every symbol* in the global context for occurrences of *sym*. (You can get a list of the names of all global symbols using `Names[Global`*]`.) You will want to use the `dependson` function that we developed in Section 7.3.7 as a starting point. Your rule can stop checking as soon as it finds any global symbol that depends on the symbol *sym*.

8.5 Additional Resources

8.5.1 Packages about packages

A skeleton package that you can fill in with your own code is in the file *Skeleton.m*, located in the *Programming Examples* subdirectory of the main *Packages* directory (file and directory names may vary on different platforms).

The collection of files entitled "Developer Tools for Applications Package Testing" (*MathSource* item #0204-961) contains documentation and packages that deal with the design and testing of other packages. This collection contains the package *EscapingSymbols.m* (discussed in Section 8.3.2), among other things.

8.5.2 Publications

[Maeder 91] contains a lucid discussion of contexts and packages.

Most of the information in Section 8.3 has been adapted from [WRI 93d] and [Gayley 94d], each of which contains many useful coding tricks and package style guidelines. Both are available from *MathSource* and are highly recommended. [WRI 93d] is one of the components of WRI's "Developer Tools for Applications Package Testing" (see above).

Packages also can be distributed as notebook files; style guidelines for these types of packages can be found in [WRI 93d], and portability issues are discussed in [Novak 94]. The *Factorization.ma* and *DependencyAnalysis.ma* packages on the supplementary diskette are written in this format. See Section 7.3.7, "Application: Dependency analysis," and Section 9.6, "Application: Defining a New Data Type," for descriptions of these packages.

9

Details, Details

When you create a quick-and-dirty function for your own immediate use, it doesn't really matter how you write it. On the other hand, if you write a function that is going to be used by other people, it should behave as much like a built-in function as possible. This chapter covers a potpourri of implementation techniques that are not widely known or used outside of the built-in functions and standard packages.

Section 9.1 discusses the proper use of diagnostic messages, and why it's a good idea to use the `Message` function for this purpose. Section 9.2 covers some advanced techniques for working with options. Section 9.3 explains the much-misunderstood numerical evaluation mechanisms of *Mathematica* and how to control them. Section 9.4 shows how to change the output formatting of expressions by attaching format values to symbols. Section 9.5 points out recommended precautions to take when you make definitions that affect the global state of a *Mathematica* session. Finally, Section 9.6 unifies a number of the techniques presented in this and earlier chapters with a case study.

This chapter is concerned mainly with minutiae. "Big picture" style guidelines, e.g., for entire packages, were discussed in Section 8.3.

9.1 Diagnostic Messages

Diagnostics should be generated using `Message`, not `Print`. In a nutshell, here is how the `Message` mechanism is used:

Define a message of the form *symbol*`::`*tag*.

```
foo::bar = "This is a message with parameters `1` and
`2`.";
```

Note that `Message` returns `Null`.

```
Message[foo::bar, "this", "that"]
foo::bar: This is a message with parameters this and
    that.
```

(Usage messages, and the issues specific to them, were discussed in Sections 8.3.2 and 8.3.3.)

9.1.1 Why use Message?

You can't pass a string as the first argument to the `Message` function — you must take the extra step of defining a message of the form *`symbol::tag`* and passing that object instead. Although this seems like unnecessary work, there are at least five good reasons for going to the trouble.

First of all, from a code maintenance standpoint, it is easier to find all of the messages that can be generated by a package if they are declared all in one place, as opposed to having strings scattered throughout the code of the entire package. Furthermore, users can get a list of all messages associated with a symbol by using `Messages[`*`symbol`*`]`.

```
Messages[foo]
{HoldPattern[foo::bar] :>
   This is a message with parameters `1` and `2`.}
```

Second, the `Message` mechanism allows a user to enable or suppress messages on an individual basis using `On[`*`symbol::tag`*`]` and `Off[`*`symbol::tag`*`]`.

```
Off[foo::bar];
Message[foo::bar, "ain't gonna", "happen"]

On[foo::bar];
Message[foo::bar, "I'm", "baaaack!"]
foo::bar: This is a message with parameters I'm and
    baaaack!.
```

The third reason to use the `Message` mechanism is that the system-defined variable `$Messages` contains a list of output streams (Section 12.3, "Low-Level Output") to which messages are written. (The default value of this variable is the list `{"stdout"}`.) The user can thus redirect all `Message` output by modifying this variable. For example, messages could be echoed to a file like this:

We use `Block` to alter the value of `$Messages` temporarily.

```
msglog = OpenWrite["msglog"];
Block[{$Messages = Append[$Messages, msglog]},
    Message[foo::bar, "echoed to", "file msglog"];
]
Close[msglog];
foo::bar: This is a message with parameters echoed to
     and file msglog.
```

`!!`*`file`* prints the contents of a file to the screen.

```
!!msglog
foo::bar: This is a message with parameters echoed to
and file msgdemo.
```

Fourth, the user can control message parameter formatting by assigning a function to the global variable `$MessagePrePrint`.

The default for `$MessagePrePrint` is `Short`.

```
Message[foo::bar, Range[1000], "that wasn't so bad"]
foo::bar: This is a message with parameters
    {1, 2, 3, 4, 5, 6, 7, <<990>>, 998, 999, 1000}
     and that wasn't so bad.
```

Here, message parameters are printed in `TeXForm`.

```
Block[{$MessagePrePrint = TeXForm},
    Message[foo::bar, a^2/b^2, Sin[theta]]
]
foo::bar: This is a message with parameters
    {{{a^2}}\over {{b^2}}} and \sin (\theta ).
```

Fifth, message generation can be detected, allowing for enhanced diagnostics or error recovery. `Check[`*`expr`*`,` *`failexpr`*`]` returns *`expr`* if no messages are generated during the execution of *`expr`*, otherwise it returns *`failexpr`*. (Additional arguments to `Check` can be used to give the names of specific messages to intercept.)

This code returns the symbol `$Failed` if a message is generated.

```
example[func_] :=
    Check[Integrate[func, {x, 1, Infinity}], $Failed]
```

In this instance the integral is evaluated successfully.

```
example[1/x^2]
1
```

In this instance the integral blows up. Since a message is generated, `Check` returns its second argument.

```
example[1/x]
Integrate::idiv:
                  1
   Integral of - does not converge on {1, Infinity}.
                  x

$Failed
```

A real program might attempt to take some sort of corrective action that depends upon the message that was generated. The global variable `$MessageList` contains a list of all messages generated during the current evaluation.

Each message in `$MessageList` is wrapped in `HoldForm`.

```
Check[Integrate[1/x, {x, 1, Infinity}],
    If[MemberQ[$MessageList, HoldForm[Integrate::idiv]],
       "the integral does not converge",
       trySomethingElse]]
Integrate::idiv:
                  1
   Integral of - does not converge on {1, Infinity}.
                  x
the integral does not converge
```

After evaluation is completed, the `$MessageList` resulting from evaluating `In[n]` is saved in `MessageList[n]`. Therefore, errors in earlier evaluations can be detected using an expression like the following:

`$Line` contains the current `In/Out` number. Thus, this returns a list of the messages generated by the *previous* evaluation.

```
MessageList[$Line-1]
{Integrate::idiv}
```

9.1.2 General messages

`Message[`*`symbol`*`::`*`tag`*`]` works by first searching `Messages[`*`symbol`*`]` for a message with tag *`tag`*. If found, that string is used.

```
Messages[foo]
{HoldPattern[foo::bar] :>
   This is a message with parameters `1` and `2`.}
```

If no such message is found, `Message` tries to use the message `General::`*`tag`*. This allows functions to share message definitions. For example,

```
General::argbu
`1` called with 1 argument; between `2` and `3`\
  arguments are expected.

Message[foo::argbu, "foo", 3, 4]
foo::argbu:
   foo called with 1 argument; between 3 and 4
     arguments are expected.
```

Use `General` messages, rather than defining new ones, whenever possible. You can get a list of all `General` messages by evaluating `Messages[General]`. Each of these messages is documented in [WRI 91].

The argument count message shown above is one of several that deal with that type of error. We should have used these system-defined messages in the `LCGRandom` example of Section 8.2.2. For example:

The actual number of arguments supplied is obtained using `Length`.

```
LCGRandom[] := "a random number"
LCGRandom[x__] :=
    Message[LCGRandom::argrx, "LCGRandom",
            Length[{x}], 0]

LCGRandom[a, b]
LCGRandom::argrx:
   LCGRandom called with 2 arguments; 0
     arguments are expected.
```

9.1.3 Issuing diagnostics without evaluating

There is one more stylistic point regarding diagnostic messages: Built-in functions return *without evaluating* when they are called incorrectly. If you have ever tried in

frustration to return an unevaluated expression from a function, you may be wondering how this can be done. In fact, it's quite easy once you know the trick: Put the `Message` command in a condition clause (`/;`). Since `Message` returns `Null`, the condition fails and the body of the rule can't execute. The message is generated, and if no other rule matches the function call does not evaluate. (This trick won't work if a function contains a catchall rule, because that rule will always end up matching.)

Get rid of the old definition of `LCGRandom[x__]` first.

```
LCGRandom[x__] =.
```

Notice the form of this definition. The body can *never* execute.

```
LCGRandom[x__]  := "never happens" /;
    Message[LCGRandom::argrx, "LCGRandom",
            Length[{x}], 0]
```

The message is printed and the function *does not evaluate.*

```
LCGRandom[a, b]
LCGRandom::argrx:
   LCGRandom called with 2 arguments; 0
     arguments are expected.
LCGRandom[a, b]
```

The flexibility of this approach is enhanced by the use of boolean combinations of the error-checking code and a call to `Message` in the rule condition. Make sure `Message` is the last part of the boolean operator. For example, the following rule for `LCGRandom` handles both the correct behavior and the erroneous behavior:

The triple `Blank` matches zero or more arguments.

```
Clear[LCGRandom]
LCGRandom[x___]  := "a random number" /;
    Length[{x}] == 0 ||
    Message[LCGRandom::argrx, "LCGRandom",
            Length[{x}], 0]
```

`Length[{x}]` is `0`, `Message` does not execute, and the rule matches.

```
LCGRandom[]
a random number
```

`Length[{x}]` is `1`, `Message` executes, and the rule does *not* match.

```
LCGRandom[3]
LCGRandom::argrx:
   LCGRandom called with 1 arguments; 0
     arguments are expected.
LCGRandom[3]
```

If `Length[{x}]` is `0` (i.e., no arguments), the second half of the rule condition does not execute, so no message is generated. On the other hand, if `Length[{x}]` is not `0`, the `Message` function *is* called, which generates the message *and* causes the rule condition to fail.

Since any expression can be used inside of a condition clause, this trick allows you to check for arbitrarily complicated error conditions before generating a message.

Exercise

1. Create definitions for the `LCGSetSeed` function of Section 8.2.2 that check not only for the correct number of arguments (one), but also for the following error condition:

This message is generated if the argument is not an integer between 1 and `LCGRandom`Private`modulus-1.

```
LCGSetSeed::badvalue = "`1` is not a valid\
  seed value; integer between `2` and `3` expected.";
```

9.2 Options

As a general rule, use options, rather than global variables, to control the way a function works. We discussed the basics of options in Section 6.3.5. Here is a recap of what was covered in that section; if any of this seems unfamiliar, you may wish to review that section before proceeding with this one.

1. Options to a function are specified as `Rule` or `RuleDelayed` objects. Options, if present, always follow nonoptional arguments.
2. The correct pattern for matching any sequence of options is `___?OptionQ`.
3. Option values are extracted using `ReplaceAll` (`/.`). In order to have user-specified option values override default option values, value extraction is done thus:

```
{optionName1, optionName2, ...} /.
    Flatten[{userSpecifiedOptions}] /. {defaultOptions}
```

A concrete example of this technique will be given below.

9.2.1 Default option values

A function `f` that supports options should allow the user to change the default option values using `SetOptions`. This is accomplished by storing default option values in `Options[f]` and evaluating `Options[f]` each time `f` is called.

`ClearAll` clears values, options, and attributes. Then set the default options for the function `f`.

```
ClearAll[f]
Options[f] = {opt1->1, opt2->2};
```

This is the canonical way to handle options with default values.

```
f[arg1_, opts___?OptionQ] :=
  Module[{opt1val, opt2val},
    {opt1val, opt2val} =
        {opt1, opt2} /. Flatten[{opts}] /. Options[f];
    {arg1, opt1val, opt2val}
  ]
```

For illustrative purposes, `f` returns a list consisting of its argument and the values it intends to use for each option. Here are some examples:

By default, the option values are 1 and 2.

```
f[x]
{x, 1, 2}
```

A manifest option overrides the default value.

```
f[x, opt2->y]
{x, 1, y}
```

SetOptions returns the new list of defaults.

```
SetOptions[f, opt2->y]
{opt1 -> 1, opt2 -> y}
```

The default for opt2 is now y.

```
f[x]
{x, 1, y}
```

The default can still be overridden.

```
f[x, opt2->z]
{x, 1, z}
```

9.2.2 Checking options for validity

Built-in functions print the General::optx message when passed an option that they don't understand.

```
General::optx
Unknown option `1` in `2`.
```

This needn't be a fatal error, but the user should be informed in case he or she simply spelled one of the options incorrectly. Finding invalid options is easy:

Suppose that this sequence of options is passed to the function f.

```
opts = Sequence[opt1->w, opt2->y, opt3->z, opt4->42]
Sequence[opt1 -> w, opt2 -> y, opt3 -> z, opt4 -> 42]
```

Here are the names of all valid options for f.

```
valid = First /@ Options[f]
{opt1, opt2}
```

Simply check that the left-hand side of each rule in opts is a member of valid.

```
MemberQ[valid, First[#]]& /@ {opts}]
{True, True, False, False}
```

Below, we integrate this code into the body of f. We use Scan to apply to the list of supplied option names a pure function that checks for option validity. If an option is invalid, the message f::optx (not General::optx) is generated.

```
Clear[f]

fcall:f[arg1_, opts___?OptionQ] :=
   Module[{opt1val, opt2val,
           valid = First /@ Options[f]},
      Scan[If[!MemberQ[valid, First[#]],
              Message[f::optx, ToString[First[#]],
                      ToString[Unevaluated[fcall]]]]]&,
           Flatten[{opts}]
      ];
```

```
    {opt1val, opt2val} =
        {opt1, opt2} /. Flatten[{opts}] /. Options[f];
    {arg1, opt1val, opt2val}
]
```

Note the use of the pattern variable `fcall` to capture the entire expression matched by the rule (see Section 6.3.6, "Assigning names to entire patterns"). This allows `ToString[Unevaluated[fcall]]` to be passed as the third argument to `Message`, making the diagnostic as meaningful as possible.

```
f[x, opt1->w, opt2->y, opt3->z, opt4->42]
f::optx: Unknown option opt3 in
    f[x, opt2 -> y, opt3 -> z, opt4 -> 42].
f::optx: Unknown option opt4 in
    f[x, opt2 -> y, opt3 -> z, opt4 -> 42].
{x, w, y}
```

9.2.3 Filtering options

Sometimes a function needs to call other functions that take options of their own. In such cases the calling function must be sure to pass to those functions only the options that they understand, or else error messages will be generated. The package `Utilities`FilterOptions`` defines a function `FilterOptions[f, opts]` that returns a sequence of the options in *opts* that are valid for *f*.

Load the package.

```
Needs["Utilities`FilterOptions`"]
```

Recall the phony options we used in the last example.

```
opts
Sequence[opt1 -> w, opt2 -> y, opt3 -> z, opt4 -> 42]
```

If we filter this sequence of options with respect to `f`, the invalid options `opt3` and `opt4` are removed.

Note that `FilterOptions` returns another `Sequence`.

```
okayOpts = FilterOptions[f, opts]
Sequence[opt1 -> w, opt2 -> y]
```

`Sequence` was discussed in detail in Section 7.1, but a brief refresher is appropriate here. `Sequence` is the head of expressions that match patterns like `x__` or `x___`. Whenever a `Sequence` appears inside of another expression, its head is stripped off and the elements of the `Sequence` are "spliced" into the enclosing expression. `FilterOptions` takes a sequence of options, such as might be matched by `___?OptionQ`, and returns a new sequence of options. Because the return value of `FilterOptions` is a `Sequence`, it can be inserted into a function call and the effect is the same as inserting the individual options.

```
f[x, okayOpts]
{x, w, y}
```

As a nontrivial example of a function that uses `FilterOptions`, we will write a function called `fitPlot` that takes a list of data points, fits a least-squares polynomial to them, and then plots both the data points and the polynomial on the same graph. The degree of the polynomial will be specified by an option. `InterpolationOrder` is an option to the built-in function `Interpolation`, so it would be consistent to use it here as well.

```
Options[fitPlot] = {InterpolationOrder->3};
```

In addition to `Interpolation` order, users may pass to `fitPlot` any options that are accepted by `Plot`, `ListPlot`, or `Graphics` (the latter are passed to `Show` to combine the outputs of `Plot` and `ListPlot`). `FilterOptions` is used to select the appropriate options in each case.

```
fitPlot[data_?MatrixQ, opts___?OptionQ] :=
Module[{order, plotOpts, listplotOpts, gfxOpts,
        f, p, lp, x, n},
```

Get `InterpolationOrder`.

```
    order = InterpolationOrder /.
                {opts} /. Options[fitPlot];
```

Filter the options.

```
    plotOpts = FilterOptions[Plot, opts];
    listplotOpts = FilterOptions[ListPlot, opts];
    gfxOpts = FilterOptions[Graphics, opts];
```

The options are printed for illustrative purposes.

```
    Print[{order, {plotOpts},
           {listplotOpts}, {gfxOpts}}];
```

Plot the data.

```
    lp = ListPlot[data, DisplayFunction->Identity,
                  listplotOpts];
```

Fit a curve.

```
    f = Fit[data, Table[x^n, {n, 0, order}], x];
```

Plot the curve-fit.

```
    p = Plot[f,
             Evaluate[{x, data[[1,1]], Last[data][[1]]}],
             DisplayFunction->Identity,
             Evaluate[plotOpts]
        ];
```

Combine the plots.

```
    Show[p, lp, DisplayFunction->$DisplayFunction,
         gfxOpts];
]
```

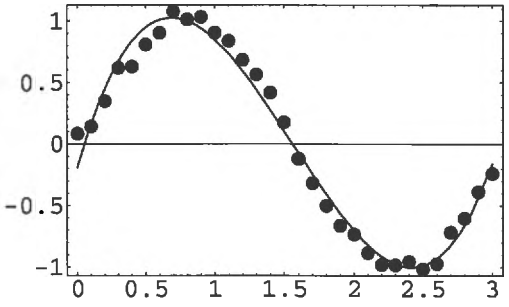
Here are some test data.

```
pts = Table[{i, Sin[2i] + (Random[]-.5)/5},
            {i, 0, 3, .1}];
```

This fits and plots a fourth-degree fit to the data. Note the different options that are passed to each function.

```
fitPlot[pts, InterpolationOrder->4,
    PlotPoints->20, Frame->True,
    PlotStyle->PointSize[.03]]
```

```
{4, {PlotPoints -> 20, Frame -> True,
   PlotStyle -> PointSize[0.03]},
   {Frame -> True, PlotStyle -> PointSize[0.03]},
   {Frame -> True}}
```



Exercise

1. Add code to `fitPlot` to check for invalid options.

9.3 Numerical Evaluation

There are many nuances of numerical evaluation, particularly in regard to the interaction between user-defined functions and the numerical evaluation operator `N`. There are several mechanisms available to make user-defined functions behave numerically as the built-in ones:

1. The arguments to a function can be tested to see whether they are exact or approximate, and different rules can be written for each case.
2. The arguments to a function can be shielded from numerical evaluation.
3. The behavior of `N[`*`expr`*`]` can be overridden.
4. Upvalues can be defined for `Series[`*`expr`*`, ...]`.

9.3.1 Numerical arguments

Many important functions can be evaluated only approximately except for certain special cases — transcendental and special functions are good examples of this. The built-in numerical functions all observe the convention of never returning an approximate result given exact inputs unless the user specifically requests it (using `N`). Your functions should observe this convention as well. If the arguments are exact and no special rule exists for those particular values, a function should return unevaluated, or perhaps rewritten in terms of other exact expressions. However, if the user supplies an approximate argument, or if the user calls `N[f[arg]]`, a function should return an approximate numerical answer.

Mathematica provides several predicates and other functions for testing the numerical properties of an expression. The simplest of these, `NumberQ`, returns `True` if its argument is a *manifest* number, i.e., has the head `Integer`, `Rational`, `Real`, or `Complex`.

```
NumberQ /@ {2, 2/5, 2.5, 2 + 5 I, 2. + 5 I, 2^(1/5)}
{True, True, True, True, True, False}
```

Note that `NumberQ[2^(1/5)]` returns `False`, because `2^(1/5)` has the head `Power`. When the intent is to determine if an expression represents a numeric, as opposed to symbolic, quantity, this behavior is undesirable. To remedy this situation, version 3.0 contains a new predicate called `NumericQ`.

```
NumericQ /@ {2.5, 2^(1/5), E^(2 + 5 I)}
{True, True, True}
```

`NumericQ` essentially answers the question, If `N` were applied to this expression, would the result be a number? This question can be answered in version 2.2 and earlier with a test like `NumberQ[N[expr]]`. However, `NumericQ` is more efficient because it is able to make this determination *without* numerically evaluating the expression (see the exercises).

Version 3.0 also adds two more new predicates, `ExactNumberQ` and `InexactNumberQ`. Like `NumberQ`, they return `True` only if their argument is a manifest number.

```
ExactNumberQ /@
    {2, 2/5, 2.5, 2 + 5 I, 2. + 5 I, 2^(1/5)}
{True, True, False, True, False, False}

InexactNumberQ /@
    {2, 2/5, 2.5, 2 + 5 I, 2. + 5 I, 2^(1/5)}
{False, False, True, False, True, False}
```

Users of versions prior to 3.0 can mimic the behavior of `ExactNumberQ` and `InexactNumberQ` using `(NumberQ[#] && Precision[#] == Infinity &)` and `(NumberQ[#] && Precision[#] < Infinity &)`, respectively.

Suppose we wanted to implement *Erlang's loss formula* (sometimes called Erlang's *B* function), which gives the probability that a customer arriving at an M/M/*c*/*c* queuing system is refused service [Jain 91]. In telephony systems analysis, `ErlangB[c, a]` is used to estimate the probability that a telephone system with `c` lines is unable to provide an open line to a user wishing to place a call, where `a` is the average load placed on the system by the (Poisson) arrival of telephone users. The formula is:

The general case.

```
ErlangB[c_Integer?Positive,
        a_?(NumberQ[#] && Positive[#]&)] :=
    (a^c/c!) / Sum[a^k/k!, {k, 0, c}]
```

A few boundary conditions.

```
ErlangB[_Integer?NonNegative, 0] = 0;
ErlangB[0, _?Positive] = 1;
ErlangB[Infinity, _?NonNegative] = 0;
```

Note that in the general case we have specified that `c` (the number of servers) must be a positive integer and that `a` (the load) must be a positive number. Each of these con-

ditions evaluates to `False` unless the corresponding argument is a manifest number.[1] For example,

```
ErlangB[3, #]& /@ {a, 2, 2., Sqrt[2]}
```

$$\{\text{ErlangB}[3, a], \frac{4}{19}, 0.210526, \text{ErlangB}[3, \text{Sqrt}[2]]\}$$

While it's certainly possible to return a symbolic result for symbolic arguments, there is little point to doing so, as it just complicates the algebraic mess with which the user probably is dealing. Furthermore, doing so would allow the user to substitute "impossible" values for `c` and `a` into the symbolic formula later on without any consistency checking.

Nevertheless, there may be times when users would like to see a symbolic expansion of the `ErlangB` function. We can let them "have their cake and eat it too" by making the following definition:

This is an upvalue for `ErlangB`.

```
Expand[ErlangB[c_, a_]] ^:=
    (a^c / c!) / Sum[a^k / k!, {k, 0, c}]
```

Symbolic expansions of Erlang's formula can now be obtained on demand.

```
Expand[ErlangB[3, Sqrt[2]]]
```

$$\frac{\text{Sqrt}[2]}{3\ (2 + \frac{4\ \text{Sqrt}[2]}{3})}$$

While there's nothing incorrect about the definition of `ErlangB` so far, there is a more efficient way to evaluate `ErlangB` numerically. The truncated exponential expansion in the denominator of Erlang's formula is equivalent to[2]

```
Sum[a^k / k!, {k, 0, c}]
```

$$\frac{(1 + c)\ E^{a}\ \text{Gamma}[1 + c, a]}{\text{Gamma}[2 + c]}$$

Evaluating this expression is more efficient than computing the explicit sum if `c` is large. However, `Gamma[1 + c, a]` does not evaluate if both `c` and `a` are exact, so we will use this method only when `a` is inexact (since `c` is constrained to be an integer). Here is the new definition:

First `Unset` the old rule.

```
ErlangB[c_Integer?Positive,
        a_?(NumberQ[#] && Positive[#]&)] =.
```

---

1. Although it may seem as though `Positive[#]` makes `NumberQ[#]` redundant, such is not the case. For example, `Positive[Sqrt[2]]` returns `True` but `NumberQ[Sqrt[2]]` returns `False`.
2. Users of versions prior to 3.0 will have to load the standard package `Algebra`SymbolicSum`` in order to simplify this sum, and the result will be a slightly different, but equivalent, expression.

These two rules replace the previous one.

```
ErlangB[c_Integer?Positive,
        a_?(ExactNumberQ[#] && Positive[#]&)] :=
    (a^c / c!) / Sum[a^k / k!, {k, 0, c}]
ErlangB[c_Integer?Positive,
        a_?(InexactNumberQ[#] && Positive[#]&)] :=
    (a^c / c!) Gamma[2 + c] /
        ((1 + c) E^a Gamma[1 + c, a])
```

```
ErlangB[3, 2.5]
```

```
0.282167
```

Compare to the exact result, which uses the explicit summation formula.

```
ErlangB[3, 5/2]
```

$$\frac{125}{443}$$

```
N[%]
```

```
0.282167
```

Exercises

1. The `NumericQ` predicate works by checking a function attribute called `NumericFunction`, which all of the built-in numeric functions in version 3.0 possess. The algorithm for `NumericQ` is simply this: Numbers are numeric, and numeric functions of numeric quantities are numeric. Implement a recursive version of `NumericQ`.
2. Implement the following diagnostic message for the `ErlangB` function:

```
ErlangB[3., 5/2]
ErlangB::nserv:
   Number of servers is expected to be a positive
     integer; found 3. instead.
```

$$\text{ErlangB}[3., \frac{5}{2}]$$

9.3.2 Shielding arguments from numerical evaluation

In most cases, handling `N[func[arg]]` is automatic, because of the way that `N` works. The argument to `N` is evaluated first in the normal way; if the result is nonnumeric, then the `N` function is applied recursively to all parts of the result (including the head, if the result is a normal expression). Finally, the `N` function is applied to the overall result.

Here is a simple example that illustrates the operation of `N`:

This function rounds inexact numbers.

```
Clear[f]
f[x_] /; Precision[x] < Infinity := Round[x]
```

The options to `Trace` are explained in Section 13.1.

```
Trace[N[f[3]], TraceInternal->True, TraceOff->f]
```

```
{N[f[3]], {f[3.], 3}, 3.}
```

Since N does not hold its arguments, `f[3]` is evaluated. In this example, `f[3]` evaluates to a normal expression (itself), so N goes to work. First, the head and argument of `f[3]` are evaluated numerically, yielding `f[3.]`. This expression evaluates according to the rule given above to produce 3 (an exact integer). The result is evaluated numerically *again*, giving `3.` (a real number) as the final answer. This strategy is designed to produce a numerical result if at all possible.

Returning to the `ErlangB` function, we see that there's a slight problem: When `ErlangB` is evaluated numerically, the first argument becomes a real number, which prevents any of the rules from matching.

```
N[ErlangB[3, Sqrt[2]]]
ErlangB[3., 1.41421]
```

Because of the upvalue for `ErlangB` defined on page 268, applying `Expand` to this expression yields a numerical answer. But that is very un-*Mathematica*-like behavior — `N[`*`anything`*`]` should return a numerical answer if at all possible!

One way to solve this problem would be to allow approximate values as the first argument to `ErlangB`. However, from a physical standpoint this makes no sense (the first argument is the number of servers in a queuing system, which must be an integer). The rules could be written so that the value of the first argument to `ErlangB` has to be within some small tolerance of the nearest integer, but that is an inelegant solution.

The best solution to the problem is to prevent numerical evaluation of the first argument to `ErlangB` in the first place. There is an attribute called `NHoldFirst` that does exactly that.[3]

```
SetAttributes[ErlangB, NHoldFirst]
```

The first argument to `ErlangB` is now unaffected by N.

```
Trace[N[ErlangB[3, Sqrt[2]]],
      TraceInternal->True, TraceOff->ErlangB]
{{{Sqrt[2], Sqrt[2]}, ErlangB[3, Sqrt[2]]},
  N[ErlangB[3, Sqrt[2]]],
  {ErlangB[3, 1.41421], 0.12132}, 0.12132}
```

`NHoldFirst` has siblings called `NHoldRest` and `NHoldAll`. Their relationships are the same as for the `HoldFirst`, `HoldRest`, `HoldAll` family of attributes.

Before leaving the ErlangB example, we note that `NumericQ` doesn't work correctly with it:

```
NumericQ[ErlangB[3, Sqrt[2]]]
False
```

3. `NHoldFirst` is new in version 3.0. In version 2.2, use the (undocumented) attribute `NProtectedFirst` instead.

If we give `ErlangB` the `NumericFunction` attribute, then the above expression will return `True`.

```
SetAttributes[ErlangB, NumericFunction]
NumericQ[ErlangB[3, Sqrt[2]]]
True
```

Unfortunately, this will also cause an expression such as `NumericQ[ErlangB[3., Sqrt[2]]]` to return `True`, even though `N[ErlangB[3., Sqrt[2]]]` does not evaluate to a number. Technically speaking, then, `ErlangB` is not a true numeric function, and perhaps it shouldn't be given this attribute. A mitigating factor in this case is that if you did Exercise 9.3.1.2, the expression `ErlangB[3., Sqrt[2]]` will cause a diagnostic message, so the user will know that something is wrong.

9.3.3 Overriding N[*expr*]

There are rare cases in which it is desirable to evaluate `N[`*`f`*`[`*`arg`*`]]` differently than *`f`*`[N[`*`arg`*`]]`. An example is provided by the `Integrate` function, which attempts to do symbolic integration even if some of its arguments are inexact. If an expression like `Integrate[`*`func, range`*`]` cannot be evaluated, then `N[Integrate[`*`func, range`*`]]` calls `NIntegrate[`*`func, range`*`]`, which is a completely different function that uses completely different algorithms than `Integrate`. Here is proof:

`Integrate` cannot do this integral. A trace reveals that `N[Integrate[...]]` calls `NIntegrate[...]`.

```
Trace[N[Integrate[x/(x + Exp[x]), {x, 0, 1}]],
      N[__] | NIntegrate[__],
      TraceInternal->True, TraceForward->True]

{N[Integrate[x/(E^x + x), {x, 0, 1}]],

  {NIntegrate[x/(E^x + x), {x, 0, 1},
    WorkingPrecision -> 16, AccuracyGoal -> Infinity,
    PrecisionGoal -> 6], 0.203039}, 0.203039}
```

This kind of behavior can be effected for any function *`f`* by defining a rule of the form "`N[`*`f`*`[...]] := ...`" or "`N[`*`f`*`[...], `*`precision_`*`] := ...`". This kind of rule is stored with *`f`*, not `N`, but it is not an upvalue: It is a type of rule called an Nvalue (Section 7.1.1).

For purposes of exposition only, suppose we were to add the following rule for the function `f` defined in the last section:

```
N[f[x_], prec_:$MachinePrecision] :=
    StringForm["precision requested =`` ", prec]
```

Now observe the difference in behavior between the following two evaluations:

```
N[f[3], 20]
precision requested = 20.

N[f[3.], 20]
3.
```

Note that `N[f[3.], 20]` uses the downvalue for `f`, not the Nvalue! The reason is that `N` evaluates its arguments before checking for Nvalues, and `f[3.]` evaluates to the integer 3. This behavior is not a bug, however; it is perfectly consistent with that of the built-in functions:

This does not give a 45-digit result.

```
N[Sin[2.], 45]
0.909297426825682
```

This does, because an internal Nvalue is used.

```
N[Sin[2], 45]
0.909297426825681695396019865911744842702254971
```

The error in each case is the user's, not the system's. If you want a high-precision result from a numerical function, you have to pass in high-precision arguments!

It's difficult to come up with an uncontrived example of using Nvalues for a function that isn't already built into *Mathematica*. In addition to the `Integrate`-`NIntegrate` duo, there are `Roots` and `NRoots`; `Solve` and `NSolve`; `DSolve` and `NDSolve`; `Sum` and `NSum`; and `Product` and `NProduct`. Note that all of these examples are *functionals* (functions that operate on other functions). For functions of numeric arguments, you can almost always effect the desired behavior simply by using different rules for exact and approximate arguments (as we did in the `ErlangB` example). In the example using the function `f` above, if there were no Nvalue for `f`, the expression `N[f[3], 20]` would be converted to `f[N[3, 20]]`, so `f` could determine the precision requested simply by checking the precision of the argument.

Remember: The purpose of Nvalues is not to distinguish between exact and approximate arguments to a function; it is to distinguish between `N[`*`f[arg]`*`]` and *`f`*`[N[`*`arg`*`]]`.

Another use of Nvalues is to define mathematical constants such as `E`, `Pi`, and so forth. Once again, all of the most useful constants are built into the system, or can be expressed in terms of the built-in constants, so it's difficult to come up with an example that isn't contrived.

The following example was used in Section 7.1.1 to illustrate Nvalues: Let `omega` denote an exact solution to the transcendental equation `omega == Exp[omega]`. We created two rules for `omega`: an upvalue for simplifying the expression `Exp[omega]` and an Nvalue for evaluating `omega` numerically:

```
Exp[omega] ^:= omega
```

```
N[omega, p_:$MachinePrecision] :=
    FindRoot[x == Exp[x], {x, 1 + I},
        AccuracyGoal->p,
        WorkingPrecision->p + 10][[1,2]]
```

Omega is thus an exact representation of a transcendental number that behaves much as built-in transcendental constants such as E and Pi, in that it simplifies symbolically when possible and evaluates numerically only when forced to:

```
omega - Exp[omega]
0

N[omega, 20] - Exp[N[omega, 20]]
0. 10^-21 + 0. 10^-20 I
```

9.3.4 Functions defined by power series

Finally, a few words on power series and functions that are defined by them. When a request is made for a numerical evaluation of such a function, it is up to the implementation of the function to figure out how many terms in the series need to be used to achieve the desired precision. For series that are "well behaved," a good strategy is to use enough terms so that the first unused term is smaller than the allowable error. For an example of coding this technique, look at the implementation of the StruveH function in the package ProgrammingExamples`Struve`. This package is described in detail in [Maeder 91].

If the convergence rate of the series is harder to check, you can let the built-in function NSum do the hard part for you. NSum takes options called TargetPrecision and WorkingPrecision that allow you to set goals for the numerical properties of the result, albeit indirectly. It is important to use the correct method of convergence checking (there are two). A discussion of them is beyond the scope of this book; see [Keiper 93a] for more information.

If you wish to define an upvalue for Series[f[x], ...], you should be aware that you can take any expression and turn it into a SeriesData object simply by adding a term of the form O[x]^*n*. For example,

This is an upvalue associated with the symbol e.

```
e /: Series[e[x_], {x_, 0, n_}] :=
    Sum[x^k / k!, {k, 0, n}] + O[x]^(n + 1)

Series[e[z], {z, 0, 5}]
                 2    3    4    5
1 + z + O[z]^6 + z  + z  + z  + z
               ---  ---  ---  ---
               2    6    24   120

Head[%]
SeriesData
```

9.4 Custom Output Formats

9.4.1 Format values

Mathematica allows you to define custom output formats for expressions by creating *format values*. A format value is defined using the `Format` function:

This rule is attached to the `BesselJ` symbol.

```
Unprotect[BesselJ];
Format[BesselJ[n_, x_]] := Subscripted[J[n]][x]
Protect[BesselJ];
```

`BesselJ` functions now print in a traditional format.

```
BesselJ[1, Pi/2]
```

$$J_1\left[\frac{Pi}{2}\right]$$

It is important to keep in mind that formatting functions such as `Subscripted` are just "wrappers" that affect only the displayed form of an expression, and not its internal value:

The expression still is a call to `BesselJ`.

```
FullForm[%]
BesselJ[n, x]
```

Format value rules are attached to the symbol for which they are defined, not to the `Format` function (much like Nvalues); that is why it was necessary to unprotect the `BesselJ` function before defining the format value above. The format values for a symbol can be listed using `FormatValues[`*`sym`*`]`:

Users of versions prior to 3.0 will see only the second of these rules.

```
FormatValues[BesselJ] // InputForm
{HoldPattern[
  MakeBoxes[BesselJ[n_, x_], FormatType_]] :>
   ToBoxes[Subscripted[J[n]][x], FormatType],
  HoldPattern[BesselJ[n_, x_]] :>
   Subscripted[J[n]][x]}
```

`Subscripted` is one of a number of formatting wrappers that produce two-dimensional output, but all of the others take sequences of individual expressions. Some examples of these functions are `Subscript`, `Superscript`, `Overscript` (version 3.0 only), `Underscript` (version 3.0 only), and `Subsuperscript` (version 3.0 only). For example,

```
Overscript[x, "_"]
```

$$\overline{x}$$

The way that `Subscript` and `Superscript` work has been changed in version 3.0. In earlier versions, the arguments to these functions were considered to be all subscripts or all superscripts, and the way to produce a superscripted variable, for example, was to

combine the variable and a `Superscript` object using the `SequenceForm` wrapper (which displays it arguments in a horizontal row without intervening spaces):

```
SequenceForm[a, Superscript[b, c]]
```

$$a^{bc}$$

In version 3.0, if there is more than one argument to `Superscript`, the first argument is considered to be the base of the superscripted form:

```
SequenceForm[a, Superscript[b, c]]
```

$$ab^{c}$$

This change in behavior is intended to make `Subscript` and `Superscript` consistent with the new formatting wrappers `Overscript`, `Underscript`, and `Subsuperscript`. Note that their behavior is unchanged if they have a single argument, so that code like the following does the same thing in either version of *Mathematica*:

```
SequenceForm[a, Superscript[b]]
```

$$a^{b}$$

Another very useful two-dimensional formatting wrapper is `ColumnForm`, which displays its arguments in a vertical stack. `SequenceForm` and `ColumnForm` can be combined with other formatting forms to create complicated two-dimensional output.

Note that the arguments to `ColumnForm` are in a list.

```
SequenceForm[a, Subscript[ColumnForm[{b, "~", c}]]]
```

$$a_{\substack{b \\ \sim \\ c}}$$

A formatting wrapper of a different kind is `StringForm`, which takes a control string and a sequence of arguments and substitutes those arguments into the control string at specified points. It is similar to the `printf` function of C or the `FORMAT` statement of Fortran. For example,

The substitution points in the control string are indicated by `` `n` ``.

```
showme[expr_, form_] :=
    StringForm["The `1` of `2` is `3`.",
        form, expr, form[expr]]

showme[Sin[x]^2, TeXForm]

The TeXForm of Sin[x]^2 is {{\sin (x)}^2}.

showme[Sin[x]^2, FortranForm]

The FortranForm of Sin[x]^2 is Sin(x)**2.
```

Note that, like other formatting wrappers, `StringForm` doesn't change the underlying expression. In particular, `StringForm` does *not* produce a string as its output, which is what you might expect:

```
FullForm[%]
StringForm["The `1` of `2` is `3`.", FortranForm,
  Power[Sin[x], 2], FortranForm[Power[Sin[x], 2]]]
```

If you want to get a string representation of the displayed form, use `ToString`. Note, however, that the string so obtained is a linear representation of the output that may be completely useless, except as fodder for some output device:

The \n is a newline character.

```
ToString[%] // FullForm
"                              2\nThe FortranForm of\
  Sin[x]  is Sin(x)**2."
```

Practical examples

The following two examples were created in response to questions that arose on the `comp.soft-sys.math.mathematica` Internet newsgroup.

The first question involved how to force exponentials with negative powers to appear in the numerator, rather than the denominator, of formatted output. The questioner objected to the excessive vertical dimensions of expressions like the following:

```
x^y Exp[-(x - a)^2 / s^2]
```

$$\frac{x^y}{E^{(-a + x)^2/s^2}}$$

It turns out to be not at all difficult to solve this problem. Note, however, that the problem arises only when the exponential is a term in a product, as opposed to standing alone. Therefore a format value for `Times` is necessary:

```
Unprotect[Times];
Format[a_ * Exp[e:(b_?Negative*c___)]] :=
    StringForm["`1` Exp[`2`]", a, e]
Protect[Times];

x^y Exp[-(x - a)^2 / s^2]
```

$$x^y \text{ Exp}[-(\frac{(-a + x)^2}{s^2})]$$

The second question concerned giving the result of `FactorInteger` an intuitively appealing output form. `FactorInteger` returns the prime factorization of an integer as a nested list:

This result is interpreted as $2^2*3^2*5^3*53^1$.

```
factors = FactorInteger[238500]
{{2, 2}, {3, 2}, {5, 3}, {53, 1}}
```

The easiest way to achieve the desired end is to create a new function of two arguments that displays its arguments as a base raised to a power, and to wrap this head around each of the sublists in the factorization.

```
Format[IntegerPower[x_, y_]] :=
    SequenceForm[x, Superscript[y]]

IntegerPower[a, b]
a^b

Apply[IntegerPower, factors, {1}]
{2^2, 3^2, 5^3, 53^1}
```

Now if we apply `Times` to this list, we'll get something that *appears* to be a product of powers:

```
factzn = Times @@ %
2^2 3^2 5^3 53^1
```

But in reality it's a product of `IntegerPower` objects, which is why it doesn't evaluate to a number.

```
InputForm[%]
Times[IntegerPower[2, 2], IntegerPower[3, 2],
  IntegerPower[5, 3], IntegerPower[53, 1]]
```

The exponent `1` is a cosmetic problem that is easy to fix:

```
Format[IntegerPower[x_, 1]] := x

factzn
2^2 3^2 5^3 53
```

9.4.2 Output formats

There are many different available output formats, such as `InputForm`, `OutputForm` (the default for version 2.2), `StandardForm` (the default for version 3.0), `TraditionalForm` (a new format in version 3.0 that mimics traditional mathematical notation), `TeXForm`, `CForm`, and `FortranForm`. `StandardForm` and `TraditionalForm` are based on a completely new *box representation* of two-dimensional input and output (see the first of the two rules for `FormatValues[BesselJ]` on page 274). The advantage of the box representation is that it can be used for two-dimensional input as well as output (cf. the string representation on page 276, which is useless as input). A discussion of these forms would require a discussion of box representation, which is outside the scope of the current volume. This topic will be addressed in a future companion volume on the *Mathematica* user interface.

The syntax `Format[expr, format] := expr2` can be used to create a formatting rule for a specific output format. For example, suppose we want to improve upon the way `Power` is formatted in `CForm`:

```
a^b // CForm
Power(a, b)
```

At the very least, this expression ought to print as `pow(a, b)`, because `pow` is the standard C library function for exponentiation. This is easy to implement:

```
Unprotect[Power];
Format[x_^y_, CForm] := pow[x, y]

a^b // CForm
pow(a, b)
```

Note that `CForm` is applied to the value returned by our format rule (as evidenced by the parentheses replacing the square brackets in `pow[x, y]`). Unfortunately, this feature can be a nuisance, as illustrated next.

The above output format for `Power` is a big improvement, but still not ideal: Many C compilers will produce error messages or, worse, incorrect code if `a` and `b` are not double-precision real numbers (C type `double`). What we'd really like to do is to produce an output of the form `pow((double)a, (double)b)`. The straightforward approach doesn't quite work, however.

```
Format[x_^y_, CForm] :=
    StringForm["pow((double)``, (double)``)", x, y]

a^b // CForm
StringForm("pow((double)``, (double)``)", a, b)
```

`CForm` is applied to the return value of the format rule, and `CForm` treats `StringForm` as just another C function! We might attempt to fix this problem by converting the output of `StringForm` to a string:

```
Format[x_^y_, CForm] :=
    ToString[
        StringForm["pow((double)``, (double)``)", x, y]]

a^b // CForm
"pow((double)a, (double)b)"
```

Unfortunately, now `CForm` insists on putting quotes around the string! Experimenting with other output format wrappers proves to be futile; the way out of this dilemma is to define the format rule to return *another* call to `Format` that uses the `OutputForm` output format:

In `OutputForm`, strings are displayed without quotes.

```
Format[x_^y_, CForm] :=
    Format[StringForm["pow((double)``, (double)``)",
                      CForm[x], CForm[y]],
           OutputForm]

a^b // CForm
pow((double)a, (double)b)
```

Note that the parameters `x` and `y` are wrapped in `CForm` before being formatted by `StringForm`. This allows expressions like the following to format correctly:

```
a^b^c // CForm
pow((double)a, (double)pow((double)b, (double)c))
```

9.5 Respect Existing Definitions

You should never modify the state of the user's session unnecessarily. There are many aspects of global state, including external rules for built-in functions, the protected status of functions, and system variables that may have been given specific values by the user.

9.5.1 Preserve existing definitions

Some "built-in" functions are actually implemented, wholly or in part, outside of the kernel! Members of the `Expand` family are good examples of this. The definition for `ExpandNumerator` in version 2.2 basically applies `Expand` to the `Numerator` of the input and divides the result by the `Denominator` of the input:


```
DownValues[ExpandNumerator]
{HoldPattern[ExpandNumerator[System`Private`input_,
     System`Private`options___]] :>
   If[Head[System`Private`input] === Plus,
    (ExpandNumerator[#1,
        System`Private`options] & ) /@
     System`Private`input,
    Expand[Numerator[System`Private`input],
      System`Private`options] /
     Denominator[System`Private`input]]}
```

(In version 3.0, the external downvalue for `ExpandNumerator` does some preprocessing of options and then calls the hidden function `System`Dump`expandNumerator`, which has an external downvalue that looks a lot like the one shown above.)

The point of this example is that you should never `Clear` a built-in function, because you might break it!

Nevertheless, it is inevitable that you will at some point find yourself in the position of having to clear one or more of your own rules for a built-in function. In such situations it is advisable to use `Unset` (Section 6.2.5, "Clearing definitions selectively") to surgically remove specific rules. Better yet, plan ahead by saving the predefined `DownValues` for a system symbol in a variable *before* defining any rules for that symbol, e.g.,

```
savedv = DownValues[systemsym];
```

Then you can use the command `DownValues[systemsym] = savedv`, rather than `Clear[systemsym]`, to start over.

9.5.2 Proper use of Protect and Unprotect

It is quite often the case that you want to override the behavior of a built-in function. The preferred way to do this is by using upvalues. Sometimes, however, there is no choice but to add downvalues to a built-in function, which requires unprotecting the function. A common mistake is to then reprotect the function without regard to what its state was before the new definitions were made.

For example, suppose you have written a package that modifies the built-in function `F`, and suppose that a user had unprotected `F` and then loaded your package. If your package outline is

```
Unprotect[F];
(* define new values for F *)
Protect[F];
```

then after loading your package, the user will find that `F` is once again protected! While this is at most a minor annoyance in an interactive session, think about what would happen if your package were loaded inside of *another* package that modified `F`:

```
Begin["otherpackage`"];
Unprotect[F];
...
```

Protects F.

```
Needs["your`package`"];
...
```

Causes an error!

```
F[..] := ...
Protect[F];
EndPackage[];
```

This package will fail to load properly because its attempted creation of a downvalue for `F` will fail.

The `Protect` and `Unprotect` functions have a feature that, if used conscientiously by all parties, avoids this problem. `Unprotect` returns a list of the symbols passed to it that were in fact protected before the call. For example:

Only one of these symbols is protected.

```
x; y; Protect[x];
```

`Unprotect` returns only the symbol that was protected.

```
Unprotect[x, y]
{x}
```

Roman Maeder suggests the following idiom for protecting and unprotecting preexisting symbols that are modified by a package [Maeder 91]:

It is necessary to `Evaluate` the argument to `Protect` because `Protect` has the `HoldAll` attribute.

```
BeginPackage["myPackage`"];
Begin["`Private`"];
protected = Unprotect[sym1, sym2, ...];
(* implementation *)
Protect[Evaluate[protected]];
End[]
EndPackage[]
```

When this package is finished loading, the protected state of each (preexisting) symbol modified by the package will be unchanged.

9.5.3 Preserving global variables

There are many global variables that affect the state of the system as a whole. These variables all begin with the $ character. Of particular interest here are those variables that contain functions that are applied at various stages in the evaluation process:

`$PreRead` — applied to every input string before it is parsed.

`$Pre` — applied to every input after it has been parsed, but before it is passed to the main evaluation loop (Section 7.1.3).

`$Post` — applied to every expression after it is evaluated but before the result is assigned to `Out[`*n*`]`.

`$PrePrint` — applied to every expression after it has been assigned to `Out[`*n*`]`, but before it is printed.

Judicious use of these functions can give the user a lot of control over the behavior of *Mathematica*. For example, one could use the following value for `$PrePrint` in order to make *Mathematica* print all approximate numbers in scientific notation:

```
$PrePrint = ScientificForm;
```

Here's an example of the effect of the definition.

```
373.4
3.734 10^2
```

As a developer, you may want to define a value for one of the *$global* functions, but in doing so you should not wipe out any value that has been defined by the user! For example, suppose that the user had defined:

```
$PrePrint = Short;
```

`Short` is now applied to every result before printing.

```
Range[100, 120, .1]
{100, 100.1, 100.2, 100.3, <<195>>, 119.9, 120.}
```

If you were to simply set `$Preprint = ScientificForm`, then the user would get a nasty surprise the next time he executed `Range[100, 200, .1]`! In order to respect the user's definition of `$PrePrint`, your code should be written thus:

The purpose of the private subcontext is to hide the symbol `oldPrePrint` from the user.

```
Begin["`Private`"];
oldPrePrint = If[ValueQ[$PrePrint],
                 $PrePrint, Identity];
$PrePrint =
    ScientificForm[oldPrePrint[#]] &;
End[];
```

Now both your definition and the user's definition will be applied.

```
Range[100, 120, .1]
```

$$\{100,\ 1.001\ 10^2,\ 1.002\ 10^2,\ \langle\langle 197\rangle\rangle,\ 1.2\ 10^2\}$$

Unfortunately, if the user decides to redefine `$PrePrint`, your definition of it will be lost.

```
$PrePrint = Short[#, 2]&;

Range[100, 200, .1]
{100, 100.1, 100.2, 100.3, 100.4, 100.5, 100.6,
  <<990>>, 199.7, 199.8, 199.9, 200.}
```

You can prevent this from happening by assigning to `$Pre` a function that evaluates the user's input and then resets `$PrePrint` to your version of it. Getting it right is quite tricky; an example of this technique can be found in the standard package `Utilities`ShowTime``, which overrides `$Pre` in order to time each evaluation automatically. This package is described in detail in [Maeder 91].

Another approach to this problem is taken by Jason Harris in his package *Aliases.m* (*MathSource* item #0206-851). This package allows a user to define *aliases*, or textual abbreviations. For example, if you load *Aliases.m* and evaluate the expression `Alias[string1, string2]`, then from that point on, *`string1`* is replaced by *`string2`* anywhere that the former occurs in the input. Harris uses `$PreRead` to accomplish this, but what is really novel is how he allows the user to continue to use `$PreRead`. The package defines a new symbol, `$NewPreRead`, that provides the same functionality as `$PreRead`, and the package aliases `"$PreRead"` to `"$NewPreRead"`. Thus, for example, if the user evaluates `$PreRead = func`, what *really* happens is `$NewPreRead = func`. The package's definition of `$PreRead` remains undisturbed, but the effect the user sees is the same as if the assignment to `$PreRead` actually had taken place!

9.6 Application: Defining a New Data Type

In this section we will take on one of the most challenging problems that any programmer can face: defining a completely new data type and making it look as though it is built in. The data type that we will define is an integer prime factorization, inspired by one of the examples discussed in Section 9.4.1. This exercise will bring together many of the techniques that have been developed in this and earlier chapters.

9.6.1 Design issues

We could define a factorization data type as a product of the `IntegerPower` objects introduced in Section 9.4.1, but there's a problem with this approach. The problem is that some factorizations would have the head `Times`, while others (those with a single prime factor) would have the head `IntegerPower`. Having two possible representations for a factorization would complicate the logic for every rule we have to write. Furthermore, it would be nontrivial to figure out if a given expression with head `Times` represented a factorization or not. A less obvious, and even more troublesome, problem is that there is no way, short of fully parsing an expression, to determine whether or not an `IntegerPower` object is part of a larger factorization. This is important, for example, in order to be able to create a rule that turns a solitary `IntegerPower[p, 1]` into the integer p (which would be an incorrect thing to do if the `IntegerPower` were part of a larger factorization). Therefore, we consider this approach to be fundamentally flawed. Instead, the data type that we develop here will have a unique head, `Factorization`.

A factorization will have this simple representation.

```
factzn = Factorization @@ FactorInteger[238500]
Factorization[{2, 2}, {3, 2}, {5, 3}, {53, 1}]
```

9.6.2 Formatting

We will construct a formatting rule for `Factorization` objects from low-level formatting primitives. Each pair `{a, b}` in the `Factorization` object will be formatted as a^b, and each pair `{a, 1}` will be formatted simply as `a`. Also, we will want a space character between every pair of factors. Therefore, we will apply the following two rules to a `Factorization` object:

```
formatrules = {
    {a_, 1} -> SequenceForm[a, " "],
    {a_, b_} -> SequenceForm[a, Superscript[b], " "]
};
```

Note the blank character after each superscript.

```
Factorization[{a, b}, {c, d}] /. formatrules
```

$$\text{Factorization}[a^{b}\ ,\ c^{d}\ ]$$

The individual parts of this object have head `SequenceForm`; therefore, if we apply `SequenceForm` and then flatten the result, we will end up with one big `SequenceForm` object:

```
Flatten[SequenceForm @@ %] // FullForm
SequenceForm[a, Superscript[b], " ", c,
  Superscript[d], " "]
```

Now the trailing blank character can be removed with `Drop`. Here, then, is the format value definition for `Factorization`:

```
Format[x_Factorization] :=
    Apply[SequenceForm, x /. formatrules
    ] // Flatten // Drop[#, -1]&
```

`Factorizations` now print in the format we have defined.

```
factzn
```

$$2^2\ 3^2\ 5^3\ 53$$

9.6.3 Overriding FactorInteger

As the next step toward integrating the `Factorization` data type into the system, we will override `FactorInteger` to produce a `Factorization` directly. First, we unprotect `FactorInteger`, being careful to save its protected status.

```
wasProtected = Unprotect[FactorInteger]
{FactorInteger}
```

Overriding `FactorInteger` brings up an interesting issue, namely that our definition for `FactorInteger` has to call the "real" `FactorInteger` to factor the integer in the first place! Without a bit of care, we could easily wind up in an infinite recursion. The uninspired way out of this dilemma would be to define a function having a different name, but we want our new data type to be integrated as seamlessly as possible into the system. Therefore, we will create a new rule for `FactorInteger`, but we must somehow ensure that our rule is not matched when we make the recursive call.

The following technique is due to Robby Villegas. A global variable,[4] `intercept`, is used inside of a rule condition to control whether or not the rule matches. When the rule does match, the rule immediately uses `Block` to set `intercept` to `False` for the duration of its execution. Thus, any nested calls to `FactorInteger` will not match this rule, because the rule condition will fail. When the rule is through executing, `Block` automatically restores the previous value of `intercept`. A hidden advantage of using `Block` for this purpose is that, even if the user aborts the computation, `intercept` will still be reset correctly.

4. Naturally, when we put these definitions into a package, this variable will be hidden inside of the package's *private subcontext* (Section 8.2.1).

```
intercept = True;
FactorInteger[x_Integer] /; intercept :=
    Block[{intercept = False},  (* prevent recursion *)
        Factorization @@ FactorInteger[x]
    ]
```

Don't forget to reprotect `FactorInteger`.

```
Protect @@ wasProtected
{FactorInteger}
```

Here's a demonstration of the new and improved `FactorInteger`.[5]

```
t = FactorInteger /@ Table[i, {i, 0, 10}]
```

$$\{0,\ 1,\ 2,\ 3,\ 2^2,\ 5,\ 2\ 3,\ 7,\ 2^3,\ 3^2,\ 2\ 5\}$$

All of these expressions have the head `Factorization`.

```
Head /@ t
{Factorization, Factorization, Factorization,
  Factorization, Factorization, Factorization,
  Factorization, Factorization, Factorization,
  Factorization, Factorization}
```

It would seem reasonable for expressions of the form `Factorization[{p, 1}]` to simplify to *p*. This is trivial to accomplish:

```
Factorization[{p_, 1}] := p

Head /@ t
{Integer, Integer, Integer, Integer, Factorization,
  Integer, Factorization, Integer, Factorization,
  Factorization, Factorization}
```

9.6.4 Expanding Factorizations

So far, so good! Now we need to create a function that turns a `Factorization` back into an integer. We already developed such a function, back in Section 5.1.3:

```
ExpandFactorization[x_Factorization] :=
    Times @@ Apply[Power, x, {1}]
ExpandFactorization[x_] := x

ExpandFactorization /@ t
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
```

Note that `ExpandFactorization` will not do anything unless its top-level argument is a `Factorization`. If the `Factorization` is an element at some lower level, nothing will happen:

5. Users of version 2.2 and earlier take note: Prior to version 3.0, `FactorInteger` acted as an identity operator on the integers `-1`, `0`, and `1`, which leads to an error message. You can fix this by making special-case definitions for these inputs.

```
ExpandFactorization[factzn + a]
```

$$a + 2^2\ 3^2\ 5^3\ 53$$

By analogy with the algebraic functions `Expand` and `ExpandAll`, we define a function called `ExpandAllFactorization` that expands factorizations wherever they may be in an expression. Surprisingly, this is trivial to write:

```
ExpandAllFactorization[x_] :=
    MapAll[ExpandFactorization, x]

ExpandAllFactorization[factzn + a]
```

$$238500 + a$$

9.6.5 Multiplication rules

The product of two factorizations should be another factorization. The way things are now, however, factorizations do not combine automatically:

```
f24 = FactorInteger[24];
f18 = FactorInteger[18];
f24 * f18
```

$$2\ 3^2\ 2^3\ 3$$

As a first step toward simplifying expressions of this type, here is an upvalue for `Factorization` with respect to `Times` that turns a product of two factorizations into a single factorization:

`Join` works on any expressions that have the same head.

```
Factorization /:
x_Factorization * y_Factorization := Join[x, y]

InputForm[f24 * f18]
```

```
Factorization[{2, 1}, {3, 2}, {2, 3}, {3, 1}]
```

Obviously, we have more work to do. Now we need a rule that combines matching prime factors inside of a factorization. This rule will be easier to write and more efficient if we give `Factorization` the `Orderless` attribute.

```
SetAttributes[Factorization, Orderless]
```

A step in the right direction.

```
InputForm[f24 * f18]
```

```
Factorization[{2, 1}, {2, 3}, {3, 1}, {3, 2}]
```

Because of the `Orderless` attribute, we know that any two matching prime factors must be consecutive, which makes combining them quite easy:

```
Factorization[h___, {a_, b_}, {a_, c_}, t___] :=
    Factorization[h, {a, b + c}, t]
```

```
f24 * f18
```

$$2^4\ 3^3$$

The `Orderless` attribute does give rise to a slight formatting quirk, however:

Although internally the power of 2 precedes the power of 3 ...

```
InputForm[f24]
Factorization[{2, 3}, {3, 1}]
```

when formatted, the 3 comes first.

```
f24
```

$$3\ 2^3$$

Depending on your point of view, this is either a bug or a feature. If you subscribe to the former view, take heart; you'll have a chance to rectify this situation in the exercises.

Now we are going to assume that if a user is calculating with a prime factorization, then she is going to want any integers that come into contact with the factorization to be factored as well. (It would be rather strange to want some integers in a product to be factored but not others, for example.) Therefore, we want to write a rule that turns the product of an integer and a factorization into a factorization. Because of the last rule we wrote, it suffices to factor the integer; then the product of the factorizations will combine automatically.

```
Factorization /:
Times[x_Factorization, y_Integer] :=
    x * FactorInteger[y]

f24 * 18
```

$$2^4\ 3^3$$

However, this rule has introduced a serious bug. This bug is particularly insidious because it involves two different functions calling *each other*, causing an infinite recursion. Anytime a factorization is multiplied by a prime number, say p, the prime will be converted first to `Factorization[{p, 1}]`, which immediately simplifies to the integer p. The process then begins again, leading to the infinite recursion.

`$IterationLimit` is temporarily reduced to avoid thousands of lines of output.

```
Block[{$IterationLimit = 20},
    Trace[3 * FactorInteger[4], Times[__]]]
$IterationLimit::itlim:
   Iteration limit of 20 exceeded.

{3 2^2, 2^2 FactorInteger[3], 2^2 3, 3 2^2,
  2^2 FactorInteger[3], 2^2 3, 3 2^2,
  2^2 FactorInteger[3], 2^2 3, 3 2^2,

... many identical lines removed ...
```

```
2^2 FactorInteger[3], 2^2 3, 3 2^2,
2^2 FactorInteger[3]}
```

The correct thing to do when a `Factorization` is multiplied by a prime p is to put a factor of `{p, 1}` inside the `Factorization` object directly, bypassing `Times`. However, there are *still* three more exceptional cases: `-1`, `0`, and `1`. All of these have a factorization (as returned by the built-in `FactorInteger`) of the form `{{x, 1}}`, yet none of them is prime. This is starting to seem too complicated; perhaps it would be best to start fresh.

```
Factorization /: Times[x_Factorization, y_Integer] =.
```

The following rule treats every integer (except `0`) in a uniform way:

```
Factorization /:
Times[x_Factorization, y_Integer /; y != 0] :=
    Factorization @@ Join[
        List @@ x,
        Block[{intercept = False}, FactorInteger[y]]
    ]
```

The way this rule works is as follows: First, it uses the *built-in* `FactorInteger` function (note the use of `Block`) to factor the integer `y`. The result is a list, not a `Factorization`. Then it converts the `Factorization` `x` to a `List`, and it passes both lists to `Join`. The combined list is then converted back to a `Factorization`, and the internal simplification rule for `Factorization` takes it the rest of the way.

This trace shows the `Join` step explicitly.

```
TracePrint[ 18 * f24, Join[__List]]
  Join[{{2, 3}, {3, 1}}, {{2, 1}, {3, 2}}]
2^4 3^3
```

Here is an example in which the multiplicative term is prime.

```
TracePrint[ 3 * f24, Join[__List]]
  Join[{{2, 3}, {3, 1}}, {{3, 1}}]
2^3 3^2
```

The case when the multiplier is `0` is handled differently. In this case, the rule does not match, and `Times` rewrites `f * 0` as `0`.

```
TracePrint[0 * f24, Unequal[__]]
  0 != 0
0
```

Exercises

1. Fix the following problem:

```
FactorInteger[-3] * FactorInteger[-4]
```

$$3\ -1^{2}\ 2^{2}$$

Be sure your fix works for any number of powers of `-1`.

2. Change the `Format` rule for `Factorization` so that the order in which the factors print mirrors the order in which they are stored internally.

9.6.6 Rules for powers

Despite our rules for multiplication, *quotients* of factorizations do not simplify:

```
f24 / f18
```

$$\frac{3\ 2^{3}}{2\ 3^{2}}$$

The reason for this behavior is that the denominator is a multiplicative term of the form `Power[Factorization[...], -1]`, which prevents it from combining with the `Factorization` in the numerator.

```
FullForm[%]
Times[Power[Factorization[List[2, 1], List[3, 2]],
   -1], Factorization[List[2, 3], List[3, 1]]]
```

Likewise, powers of `Factorizations` do not simplify, and furthermore they look terrible:

The apparent exponent of 32 actually is the textual concatenation of the exponents 3 and 2.

```
f24 ^ 2
```

$$3\ 2^{32}$$

It's quite easy to write a rule to take care of these problems.

```
Factorization /:
    Power[a_Factorization, b_Integer | b_Rational] :=
        a /. {x_, y_} -> {x, y b}

{ f24 / f18, f24 ^ 2, Sqrt[f24]}
```

$$\{2^{2}\ 3^{-1},\ 2^{6}\ 3^{2},\ 2^{3/2}\ 3^{1/2}\}$$

```
1 / %
```

$$\{3\ 2^{-2},\ 2^{-6}\ 3^{-2},\ 2^{-(3/2)}\ 3^{-(1/2)}\}$$

Incidentally, the parentheses in the exponents of the last two results were inserted automatically by the output formatter, because the precedence of unary minus is greater than the precedence of multiplication. For a complete discussion of how precedence

affects formatting see `PrecedenceForm` in [Wolfram 91] §2.7.8 or [Wolfram 96] §2.8.16.

There are still a few glitches left, which will be fixed in the next exercise.

Exercise

1. Fix the following problems:

```
{ f24 / 3, f24 * 2/3, r = 2/3; f24 * r }
```

$$\left\{\frac{3\ 2^3}{3}, \frac{2\ 3\ 2^3}{3}, \frac{2\ 3\ 2^3}{3}\right\}$$

```
{ f18 / f24^2, t = 1/f24; f24 * t }
```

$$\{2^{-5}\ 3^0,\ 2^0\ 3^0\}$$

9.6.7 Addition rules

Finally, here are some rules for addition of factorizations. The straightforward way to implement addition would be with a rule like this:

```
Plus[a_Factorization, b_Factorization] :=
    FactorInteger[
        ExpandFactorization[a] + ExpandFactorization[b]
    ]
```

This rule relies on the `Flat`ness of `Plus` to handle longer sums. However, this rule would be very inefficient for sums of many terms, because of all the conversions back and forth between factored and unfactored forms. Instead, we would like to write a rule that will handle a sum of any number of factorizations all in "one fell swoop."

Toward this end it will prove convenient to add a rule to `ExpandFactorization` that takes a `Sequence` of factorizations as an argument and returns a `Sequence` of integers as a result.

```
ExpandFactorization[x__Factorization] :=
    Sequence @@ ExpandFactorization /@ {x}
```

Here's what this rule does.

```
ExpandFactorization[f24, f18]
Sequence[24, 18]
```

And here's what we'll do with the result.

```
Plus[%]
42
```

Now it should be quite clear how to write a rule for `Plus` that handles any number of factorization arguments at once:

```
Factorization /:
Plus[a__Factorization] :=
    FactorInteger[Plus[ExpandFactorization[a]]]
```

Now the sum of two factorizations is another factorization.

```
f24 + f18
2 3 7
```

While we're at it, we ought to define a rule for the sum of an integer and an arbitrary number of factorizations. (It is not necessary to handle multiple integer arguments, since `Plus` will combine those on its own.) Note how we exploit the "sequenceability" of `ExpandFactorization` by passing all of the `Factorization` arguments to it at once.

```
Factorization /:
Plus[a_Integer, b__Factorization] :=
    FactorInteger[Plus[a, ExpandFactorization[b]]]

f24 + 10 + f18 + 3
5 11
```

Exercise

1. Add rules for `Log`, `Quotient`, `Mod`, the relational operators, and anything else you can think of to make them work with factorizations as arguments. Try not to expand the factorizations unless it would lead to simplification of the expression.

9.6.8 Nvalues

`Factorization` provides an uncontrived illustration of the use of Nvalues. When `N` is applied to a `Factorization`, the result should be a number, but it's not:

```
N[FactorInteger[8]]
2.^3.
```

To recast the discussion of Section 9.3.3 into the present context: Since `Factorization[{2, 3}]` does not evaluate to a numeric quantity, `N` applies itself to all parts of that expression, which results in the intermediate form `Factorization[{2., 3.}]`. `N` then attempts to evaluate this new expression, which still evaluates to itself.

We can solve this problem by defining an Nvalue that expands a factorization into a product of powers when `N` is applied. After expansion, `N` applies itself again, resulting in a numerical answer.

```
N[a_Factorization, _] := ExpandFactorization[a]

N[FactorInteger[8]]
8.
```

Note that we can ignore the precision argument because the *second* application of `N` does all the numerical work.

```
1 / FactorInteger[9]
```

$$3^{-2}$$

```
N[%, 30]
0.111111111111111111111111111111
```

The Nvalue allows us to pass factorizations to any function that numerically evaluates its arguments, such as `FindRoot` or any of the plotting functions.

```
f[x_] = x^2 - FactorInteger[56]
```

$$x^2 + -1\ 7\ 2^3$$

```
FindRoot[f[x] == 0, {x, 1}]
{x -> 7.48331}
```

9.6.9 Modifying $Pre

Suppose we want to set things up so that anytime the user entered a manifest prime raised to a manifest integer exponent, the expression would be converted to a `Factorization`. We could effect this behavior by applying a rule of the form `Power[a_?PrimeQ, b_Integer] -> Factorization[{a, b}]` to every user input *before* it evaluates.

Here is the function that we will use. Note that the function must hold its argument, or else the powers would simplify before it had a chance to act on them.

```
SetAttributes[AutoFactor, HoldFirst]
AutoFactor[command_] :=
    Unevaluated[command] /.
        a_?PrimeQ ^ b_Integer -> Factorization[{a, b}]

AutoFactor[g[2^3, 3^2 + 1]]
```

$$g[2^3, 2\ 5]$$

The question is, How do we apply `AutoFactor` to every user input? We accomplish this by assigning `AutoFactor` to the global variable `$Pre` (Section 9.5.3).

Before assigning to `$Pre` ...

```
g[2^3, 3^2 + 1]
g[8, 10]

$Pre = AutoFactor;
```

and after.

```
g[2^3, 3^2 + 1]
```

$$g[2^3, 2\ 5]$$

Naturally, production-quality code would save the user's value of `$Pre` before making this assignment.

Exercise

1. Create a package called *Factorization.m* that implements the `Factorization` data type.

9.7 Additional Resources

Mark Sofroniou has written a package called *Format.m* that provides a totally rewritten set of formatting primitives for C, Fortran, TeX, and Maple. This package is available from *MathSource* as item #0205–254.

Part 4
Programming for Performance

10

Performance Tuning

For many users the principal drawback of *Mathematica* is its lack of speed relative to other programming tools. Being an interpreted language, *Mathematica* is of course slower than compiled languages such as C or Fortran. The very features that make *Mathematica* so powerful — dynamically typed data, pattern matching, and the very general way in which expressions are evaluated (Chapter 7, "Expression Evaluation") — also exact a toll on performance.

Most of the time, however, what is perceived to be a problem with *Mathematica* actually is a problem with the way that the code is written; a different solution in *Mathematica* may be many times faster, and certain built-in functions rival the speed of compiled languages. In this chapter we examine the intricate details of performance-tuning *Mathematica* code.

There are several general rules of thumb. Some of them are obvious, such as using special-purpose built-in functions whenever possible; others are quite unobvious, such as avoiding in-place modification of large lists. The *what* of most of them can be arrived at through experimentation, but the *why* sometimes requires a deep understanding of how *Mathematica* evaluates expressions.

Some of the techniques for tuning *Mathematica* code basically are "tweaks" that speed up a computation by a constant factor; others can result in qualitatively different time complexities — for example, reducing a computation's time complexity from quadratic in the problem size to linear. Simply incorporating a few new programming idioms into the code may be all that is necessary.

Another method for speeding up a program is to use the internal compilation facility. The speedup gained through compilation can be substantial, and the technique is quite easy to use (modulo a few "gotchas," of course). Unfortunately, the compiler can handle only a limited set of *Mathematica* expressions in version 2.2; the capabilities of the compiler are greatly expanded in version 3.0.

Finally, when every last second counts, the *MathLink* protocol [WRI 93c] can be used to communicate with external programs written in compiled languages such as C or Fortran. This may seem like a lot of trouble, but in practice it is often just the opposite, as it may be the case that the C or Fortran code to solve the problem already has been written. *MathLink* is important enough (and involved enough!) to merit its own chapter (Chapter 11).

10.1 Rules of Thumb

There are many general strategies, or "rules of thumb," of which one needs to be aware. Most of these rules can be arrived at through experimentation, but to really understand them requires an understanding of the evaluation process (Chapter 7). In this section we'll list the rules and give examples to motivate them. In later sections we'll examine techniques for avoiding some of the pitfalls pointed out here.

Incidentally, the rules here are *not* necessarily presented in order of their impact on performance, but rather in the order that makes the presentation as coherent as possible.

10.1.1 Rule 1: Use built-in functions

This rule is not surprising, since built-in functions are (with a few exceptions) implemented in the kernel, which is written in C. There are various overheads involved in evaluating general expressions (see Chapter 7) that are not a factor inside of an internal function. In general, pick the method that moves as much of the computation as possible into the internal code.

As a simple example, suppose we wish to sum the squares of the first 5000 integers. This could be done quite easily using any of the looping constructs.

```
{s, i} = {0, 1};
While[i <= 5000, s += i^2; i++] // Timing
s
{3.26667 Second, Null}

41679167500
```

For no obvious reason, a `For` loop is a bit faster than an equivalent `While` loop.

```
s = 0;
For[i = 1, i <= 5000, i++, s += i^2] // Timing
{2.98333 Second, Null}
```

A `Do` loop is quite a bit faster than either of those.

```
s = 0;
Do[s += i^2, {i, 5000}] // Timing
{2. Second, Null}
```

None can match the speed of a special-purpose function like `Sum`, however.

```
Sum[i^2, {i, 5000}] // Timing
{0.933333 Second, 41679167500}
```

These examples reinforce the reasoning given above: The reason the `Do` loop is faster than the other two types of loops is probably due to the fact that all of the index variable computation is handled internally. `Sum` is faster still, since not only the index variable computation but also the addition operations are implicit rather than explicit.

10.1.2 Rule 2: Program functionally

The basic goal when using functional programming techniques to improve performance is this: Try to apply operations to as much of the data at one time as possible. (This rule is just another facet of the general strategy of trying to push as much of the computation as possible into the kernel's internal routines.) This strategy is facilitated by the fact that all numerical functions are listable (Section 3.3), and for those functions that aren't, `Map`, `Apply`, etc. can be used.

Therefore, a good general approach to the example introduced in the previous section is to generate a list of all of the summands, and then apply `Plus` to it. There are several different ways to generate the list of summands, and a little experimentation shows their relative speeds.

Using `Table` to generate the list of squares is probably the most obvious method.

```
Plus @@ Table[i^2, {i, 5000}] // Timing
{0.9 Second, 41679167500}
```

The effect of squaring the entire list at once is quite noticeable.

```
Plus @@ (Table[i, {i, 5000}]^2) // Timing
{0.633333 Second, 41679167500}
```

A further improvement comes from replacing `Table` with `Range`.

```
Plus @@ (Range[5000]^2) // Timing
{0.55 Second, 41679167500}
```

Note again that the more specific operator (`Range`) is more efficient than the more general operator (`Table`) because more of the computation is done inside of the kernel.

Of course, the strategy suggested here is trading memory for time. When there is not enough memory available to hold all of the intermediate results at one time, a procedural solution may be a better alternative. Furthermore, other factors such as the use of virtual memory can affect the performance of functional programming algorithms significantly.

10.1.3 Rule 3: Use machine-precision arithmetic

When the eventual answer is going to be an approximate number, and machine-precision arithmetic does not pose numerical problems, convert numbers to machine precision as early as possible in the computation.

Let us change the example problem so that instead of summing squares, we sum square roots. A sum of 5000 symbolic square roots is of dubious value; we almost cer-

tainly want a numerical answer. The *worst* possible way to do it would be something like this:

This would be even slower had we used a loop.

```
N[Plus @@ Sqrt[Range[5000]]] // Timing
{13.3167 Second, 235737.}
```

Here is a smaller example that demonstrates why this computation is so slow:

This builds up a large list of integers and symbolic square roots.

```
Sqrt[Range[10]]
{1, Sqrt[2], Sqrt[3], 2, Sqrt[5], Sqrt[6], Sqrt[7],
  2 Sqrt[2], 3, Sqrt[10]}
```

Many of these terms cannot be combined by `Plus`.

```
Plus @@ %
6 + 3 Sqrt[2] + Sqrt[3] + Sqrt[5] + Sqrt[6] +
  Sqrt[7] + Sqrt[10]
```

After each term is approximated numerically, `Plus` executes a *second* time.

```
N[%]
22.4683
```

Improving this code is as easy as falling off of the proverbial log. Here are some examples:

A small but noticeable improvement is effected by applying `N` to the list of square roots before passing it to `Plus`.

```
Plus @@ N[Sqrt[Range[5000]]] // Timing
{11.9833 Second, 235737.}
```

Observe the dramatic improvement when the list of integers is evaluated numerically *before* square roots are taken.

```
Plus @@ Sqrt[N[Range[5000]]] // Timing
{1.01667 Second, 235737.}
```

Machine-precision `Range` bounds obviate the need for `N` altogether!

```
Plus @@ Sqrt[Range[1., 5000.]] // Timing
{1. Second, 235737.}
```

Another small improvement comes from specifying the increment in machine precision as well.

```
Plus @@ Sqrt[Range[1., 5000., 1.]] // Timing
{0.933333 Second, 235737.}
```

Explicit loops can be sped up in exactly the same ways.

10.1.4 Rule 4: Evaluate when possible

Functions such as `For`, `Sum`, and `Plot` evaluate their first argument numerically at several values of an independent variable. Such arguments are always *held* (that is, not evaluated before the function is called):

HoldAll indicates that none of the arguments to Sum are evaluated.

```
Attributes[Sum]
{HoldAll, Protected, ReadProtected}
```

This is done under the assumption that symbolic evaluation of the expression may be incorrect until numeric values of the independent variable are substituted (e.g., if the expression contains conditional operations such as `If`).

Sometimes, however, evaluating the first argument before the call can result in an algebraic simplification that makes subsequent evaluations more efficient. Here is an example in which a seemingly trivial expression actually is far from optimal:

```
Sum[1/i^2, {i, 1., 5000., 1.}] // Timing
{1.11667 Second, 1.64473}}

Clear[i]
Sum[Evaluate[1/i^2], {i, 1., 5000., 1.}] // Timing
{0.7 Second, 1.64473}
```

What is going on here? The parser converts the input `1/i^2` to this expression:

```
FullForm[Hold[1/i^2]]
Hold[Times[1, Power[Power[i, 2], -1]]]
```

Clearly, this is a very inefficient way to evaluate `1/i^2`! If this expression is forced to evaluate, however, it assumes a much more compact and efficient form:

```
FullForm[1/i^2]
Power[i, -2]
```

It should be pointed out that for this particular sum, NSum is the fastest method by a wide margin:[1]

```
NSum[i^(-2), {i, 5000}] // Timing
{0.366667 Second, 1.64473}
```

This is because NSum checks the first few partial sums for evidence of convergence, and if it determines that the sum is converging, NSum estimates the value of the tail of the series rather than computing it explicitly.[2] The estimation methods used by NSum are discussed in [Keiper 93a].

1. *Caveat:* The first time NSum is called, it may be quite slow, as the kernel loads the code that implements it. Subsequent calls to this function are much faster.
2. Because of this fact, NSum can even be used to sum infinite series.

10.1.5 Rule 5: Compile when possible

Another technique that NSum uses is to *compile* the summand into *pseudocode* before evaluating the sum.[3] The pseudocode is basically an assembly language program for an idealized register machine that is implemented by the kernel. This pseudocode can be evaluated much more quickly than the equivalent normal expression.

Compilation is directly available to the programmer as well. Note the speed increase achieved by compiling a simple expression like `(i - 1)/(i + 1)`:

```
Sum[(i - 1.)/(i + 1.), {i, 1., 5000., 1.}] // Timing
{1.23333 Second, 4983.81}
```

The syntax for `Compile` is similar to the syntax for `Function`.

```
cf = Compile[{x}, (x - 1.)/(x + 1.)]
CompiledFunction[{x}, (x - 1.)/(x + 1.), -CompiledCode-]

Sum[cf[i], {i, 1., 5000., 1.}] // Timing
{0.916667 Second, 4983.81}
```

The speedup can be even more dramatic when the compiled function is less trivial. Not all *Mathematica* expressions can be compiled, unfortunately. We'll discuss the subtleties of compilation in Section 10.5.

10.1.6 Rule 6: Avoid Append and Prepend

This is one of the most important rules. `Append`, `Prepend`, and their ilk, when they appear inside of a loop, are like a death sentence for performance purposes. Consider the following two methods for generating a list of 1000 numbers:

```
Table[i, {i, 1000}]; // Timing
{0.0333333 Second, Null}

s = {};
Do[AppendTo[s, i], {i, 1000}]; // Timing
{2.11667 Second, Null}
```

This is an example in which the blame could mistakenly be placed on the looping construct. In fact, the loop itself is quite speedy:

```
Do[x = i, {i, 1000}]; // Timing
{0.05 Second, Null}
```

3. Many other functions (e.g., `Plot`) also compile some of their arguments. You can find out which ones do so by searching for functions having the `Compiled` option (using the techniques of Section 7.3.5).

The real problem is that functions like AppendTo that change the length of a list have to *copy* the list. In the example given, the list is copied 1000 times. When you build up a list of length n using Append or AppendTo, the number of items copied is approximately $n^2/2$.

This same phenomenon can make recursive functions that build up their result one element at a time very slow. This often puzzles Lisp programmers, who are taught to program that way because it is efficient in Lisp.

There are several solutions to this problem, which we will discuss in Sections 10.2.1 and 10.3.1.

10.1.7 Rule 7: Do not modify large lists in place

In-place modification of large lists is extremely inefficient in *Mathematica*. As an example, consider the following two ways of computing the first moving average of a list of numbers:

Here the result is computed in place (i.e., it replaces the original data).

```
s = Range[1000];
Do[s[[i]] = (s[[i]] + s[[i + 1]])/2,
   {i, Length[s] - 1}]; // Timing
{3.01667 Second, Null}
```

Here the result is stored in a different list than the original data. The creation of the destination list is included in the timing.

```
s = Range[1000];
(t = Table[0, {1000}];
 Do[t[[i]] = (s[[i]] + s[[i + 1]])/2,
    {i, Length[s] - 1}];) // Timing
{0.633333 Second, Null}
```

In Section 7.3.3 we explained that this disparity is due to the way that the kernel evaluates an expression like s[[i]] if s has been modified since its last access. The evaluation process requires a scan of each element of s to check for upvalues, which takes time proportional to the length of s (review Section 7.3.3 for a more in-depth explanation). Thus, in the first algorithm s undergoes a scan on each loop iteration, which makes the running time *quadratic* in the length of the list, rather than linear.

Unfortunately, there are three possible drawbacks to the second approach: First, the length of the result may not be known in advance; second, it may result in a lot of unnecessary data movement; and third, the list may be just too large to copy. There are other solutions to this problem, and they will be discussed in Section 10.2.2.

10.1.8 Rule 8: Beware of inefficient patterns

Pattern matching and rule replacement often provide some of the most elegant solutions to a particular problem. However, incorrect use of these techniques can drastically impair performance.

For example, the following algorithm for run-length encoding a list of data (due to Frank Zizza) is unquestionably elegant — in fact, it won an award in a programming contest at a 1990 *Mathematica* conference.

```
runEncode[s_List] :=
    Map[{#, 1}&, s] //.
        {h___, {x_, n_}, {x_, m_}, t___} :>
            {h, {x, n + m}, t}

runEncode[{1, 0, 0, 1, 1, 1}]
{{1, 1}, {0, 2}, {1, 3}}
```

The `Map` statement converts `{s1, s2, ...}` to `{{s1, 1}, {s2, 1}, ...}`; subsequently, the pattern combines adjacent sublists that have the same first element, adding their repetition counts together. Unfortunately, the running time of this algorithm appears to be a quadratic function of the input size:

Each doubling of the input size causes the run time to quadruple (at least).

```
data = Table[Random[Integer, {0, 1}], {256}];
{#, Timing[runEncode[Take[data,#]]][[1,1]]}& /@
    {32, 64, 128, 256}
{{32, 0.0166667}, {64, 0.0833333}, {128, 0.433333},
  {256, 2.31667}}
```

The problem with this approach has to do with the way the kernel performs pattern matching. We can see pattern matching in action by "instrumenting" the replacement rule with a condition clause that prints each attempted pattern match:

We use the condition clause to check `x == y` (rather than specifying `x_` twice in the pattern) to force the kernel to show us every match that it tries.

```
{{1, 1}, {0, 1}, {0, 1}, {1, 1}, {1, 1}, {1, 1}} //.
    {h___, {x_, n_}, {y_, m_}, t___} /;
        (Print[{h, {{x, n}, {y, m}}, t}]; x == y) :>
            {h, {x, n + m}, t}
{{{1, 1}, {0, 1}}, {0, 1}}, {1, 1}, {1, 1}, {1, 1}}
{{1, 1}, {{0, 1}, {0, 1}}, {1, 1}, {1, 1}, {1, 1}}
{{{1, 1}, {0, 2}}, {1, 1}, {1, 1}, {1, 1}}
{{1, 1}, {{0, 2}, {1, 1}}, {1, 1}, {1, 1}}
{{1, 1}, {0, 2}, {{1, 1}, {1, 1}}, {1, 1}}
{{{1, 1}, {0, 2}}, {1, 2}, {1, 1}}
{{1, 1}, {{0, 2}, {1, 2}}, {1, 1}}
{{1, 1}, {0, 2}, {{1, 2}, {1, 1}}}
{{{1, 1}, {0, 2}}, {1, 3}}
{{1, 1}, {{0, 2}, {1, 3}}}
{{1, 1}, {0, 2}, {1, 3}}
```

Two important facts can be gleaned from this example. First, the kernel attempts to match sequence patterns (e.g., "__", "___", "..", or "...") from left to right. Second, after the kernel finds a successful match for the pattern, on the next iteration it starts its search back at the left end of the input, which for this particular algorithm is guaranteed to be a waste of time. It would be much more efficient if pattern matching could pick up "where it left off" on the previous iteration. A rule-based technique with exactly this behavior will be discussed in Section 10.4.

10.1.9 Rule 9: Use #, &-style pure functions

This is completely unobvious, but a pure function defined using # and & is faster than a "normal" function.[4] Surprisingly, however, a definition using `Function` is about as slow as a normal function. Here is an example:

```
func1[x_] := x^-2
func2 = Function[{x}, x^-2];
func3 = #^-2 &;

z = Range[5000.];
func1 /@ z; // Timing
{2.01667 Second, Null}

func2 /@ z; // Timing
{2.08333 Second, Null}

func3 /@ z; // Timing
{1.25 Second, Null}
```

There are several things that might explain why the #, & form is faster than the normal function: no pattern matching is taking place, there is no search for upvalues or attributes, etc. However, the relative slowness of the `Function[`*`params, body`*`]` form is a mystery.

10.2 Procedural Perils

A handful of terribly inefficient procedural programming techniques are what give procedural programming in *Mathematica* a bad reputation. A handful of new procedural programming idioms are all that the programmer needs to avoid these problems.

10.2.1 Building up large lists

The problem with building up lists an element at a time is that *Mathematica* lists are implemented as arrays. The advantage of this implementation is that a list can be randomly indexed in time that is independent of the length of the list; the disadvantage is that every time an element is added to or taken away from a list, the list has to be copied into a new list — leading to a quadratic time complexity for algorithms that build up large lists one element at a time.

Consider the problem of writing a function that merges two sorted lists, which is an essential component of a *mergesort* algorithm (Section 5.4.2). Here is a (bad) procedural algorithm for doing this:

4. Allan Hayes first pointed this out to the author.

The first `While` does the merging, after which the second and third `While`'s check for "leftovers" in one list or the other.

```
pmerge1[list1_List, list2_List] :=
Module[{ i1 = 1, i2 = 1, temp = {} },
    While[i1 <= Length[list1] && i2 <= Length[list2],
        If[list1[[i1]] < list2[[i2]],
            AppendTo[temp, list1[[i1]]]; i1++,
            AppendTo[temp, list2[[i2]]]; i2++
        ]
    ];
    While[i1 <= Length[list1],
          AppendTo[temp, list1[[i1]]]; i1++ ];
    While[i2 <= Length[list2],
          AppendTo[temp, list2[[i2]]]; i2++ ];
    temp
]
```

Some test data.

```
odds = Range[1, 1999, 2];
evens = Range[2, 2000, 2];
```

`pmerge1` is quite slow.

```
Short[pmerge1[odds, evens]] // Timing
{8.5 Second, {1, 2, 3, 4, 5, 6, 7, <<1990>>,
   1998, 1999, 2000}}
```

Below, we discuss several ways to speed up this function.

Preallocate the result

The simplest solution to the problem is to preallocate a list of the appropriate size and then fill it in:

In this algorithm, the result is stored in `temp`. The length of `temp` never changes.

```
pmerge2[list1_List, list2_List] :=
Module[{ i1 = 1, i2 = 1, temp },
    temp = Table[0, {Length[list1] + Length[list2]}];
    While[i1 <= Length[list1] && i2 <= Length[list2],
        If[list1[[i1]] < list2[[i2]],
            temp[[i1 + i2 - 1]] = list1[[i1]]; i1++,
            temp[[i1 + i2 - 1]] = list2[[i2]]; i2++
        ]
    ];
    While[i1 <= Length[list1],
          temp[[i1 + i2 - 1]] = list1[[i1]]; i1++ ];
    While[i2 <= Length[list2],
          temp[[i1 + i2 - 1]] = list2[[i2]]; i2++ ];
    temp
]
```

`pmerge2` is over five times faster than `pmerge1` on the test data.

```
Short[pmerge2[odds, evens]] // Timing
{1.75 Second, {1, 2, 3, 4, 5, 6, 7, <<1990>>,
   1998, 1999, 2000}}
```

An increase in speed of a factor of 5 is impressive enough, but the real story here is that the two algorithms have different *computational complexities;* that is, their timing

behavior is qualitatively different as the size of the problem increases. Shown below are some timings of the `pmerge` functions on inputs of various sizes.

The strange-looking time values are a result of averaging several runs of each test case.

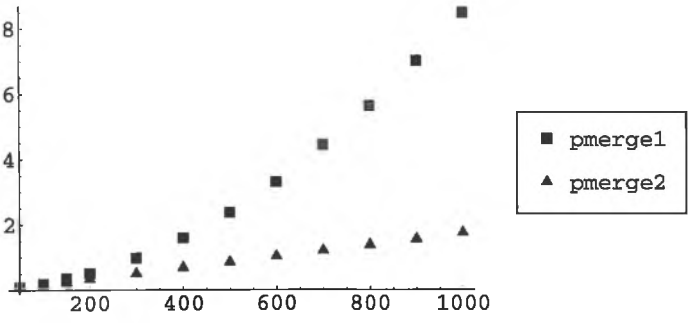
```
pmtimes1 =
 {{50, 0.0916667}, {100, 0.191667}, {150, 0.35},
  {200, 0.508333}, {300, 0.975}, {400, 1.59167},
  {500, 2.35833}, {600, 3.29167}, {700, 4.425},
  {800, 5.61667}, {900, 7.}, {1000, 8.475}};

pmtimes2 =
 {{50, 0.0916667}, {100, 0.166667}, {150, 0.241667},
  {200, 0.341667},{300, 0.508333}, {400, 0.683333},
  {500, 0.85}, {600, 1.025}, {700, 1.19167},
  {800, 1.375}, {900, 1.55}, {1000, 1.74167}};
```

The standard package `Graphics`MultipleListPlot`` is nice for plotting several sets of discrete data.

```
Needs["Graphics`MultipleListPlot`"]
```

The `SymbolShape` option is new to version 3.0; users of earlier versions can omit this option. The code to create the legend is not shown.

```
pmlp = MultipleListPlot[pmtimes1, pmtimes2,
    SymbolShape->{PlotSymbol[Box],
                  PlotSymbol[Triangle]},
    PlotRange->All];
```

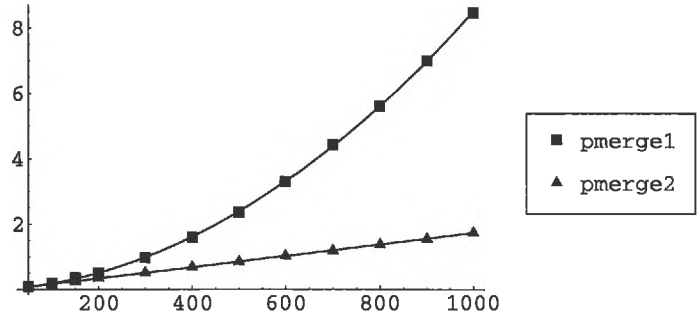


Next, we fit some least-squares polynomials to the data sets. We do not specify a constant in the list of basis functions, because in theory, each algorithm should take 0 time on an input of length 0. Note the relative strengths of the linear and quadratic terms in the two fitting functions (keep in mind that the `x` values are in the range 10^2 to 10^3).

```
Clear[x]
pmfit1[x_] = Fit[pmtimes1, {x, x^2}, {x}]

0.00104166 x + 7.45936 10^-6 x^2

pmfit2[x_] = Fit[pmtimes2, {x, x^2}, {x}]

0.00166503 x + 6.9512 10^-8 x^2
```

The following graph shows that the quadratic component of `pmerge1`'s running time is the dominant one, whereas in the case of `pmerge2` the quadratic component probably is nonexistent.

```
Show[pmlp,
    Plot[{pmfit1[x], pmfit2[x]}, {x, 50, 1000},
        DisplayFunction->Identity],
    DisplayFunction->$DisplayFunction,
    PlotRange->All
];
```



Use linked lists

The preallocation strategy may be fine when the size of the result is known in advance, but what about cases in which it is not? In such cases, list nesting can be used to implement a *linked-list* data structure. Here is the basic idea:

```
s = {};
Do[s = {s, x}, {x, 1, 5}]; s
{{{{{{}, 1}, 2}, 3}, 4}, 5}
```

What we have here is a list whose second element is the number 5 and whose first element is another list. The second list also contains two elements, the number 4 and another list, and so on. No list's size is ever changed during this process. A new list (of two elements) is created at each step,[5] but since that is a constant-time operation, the entire cost is linear in the number of iterations. At the end of this process, the nested list can be turned into a one-level list by flattening it, which is another linear-time operation:

```
Flatten[%]
{1, 2, 3, 4, 5}
```

Here is a procedural merge function that builds up its result as a linked list:

5. Lisp programmers will recognize this as a `cons` operation.

`temp[[i1 + i2 - 1]] = list1[[i1]]` has been replaced by `temp = {temp, list[[i1]]}`, etc.

```
pmerge3[list1_List, list2_List] :=
Module[{ i1 = 1, i2 = 1, temp = {} },
    While[i1 <= Length[list1] && i2 <= Length[list2],
        If[list1[[i1]] < list2[[i2]],
            temp = {temp, list1[[i1]]}; i1++,
            temp = {temp, list2[[i2]]}; i2++
        ]
    ];
    While[i1 <= Length[list1],
          temp = {temp, list1[[i1]]}; i1++ ];
    While[i2 <= Length[list2],
          temp = {temp, list2[[i2]]}; i2++ ];
    Flatten[temp]
]

pmerge3[{1, 3, 4}, {2, 5}]
{1, 2, 3, 4, 5}
```

Figure 10-1 graphically compares the running time of `pmerge3` with that of `pmerge1` and `pmerge2`. (The code for generating this and all subsequent graphics has been omitted to save space. The size of the input is on the horizontal axis, and the running time in seconds is on the vertical axis.) As promised, `pmerge3` has linear time complexity, and in addition it turns out to be faster than `pmerge2`. This probably is due to the index calculations required in `pmerge2`.

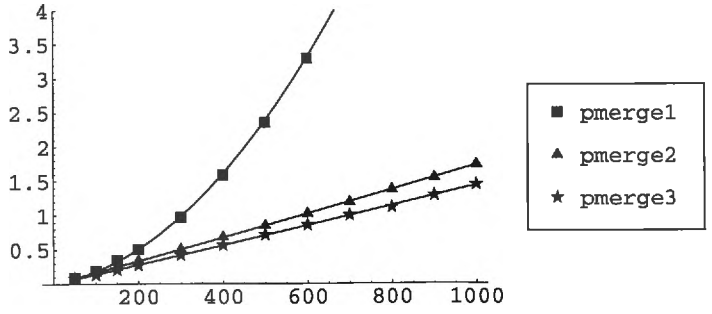


Figure 10-1

The only fly in this ointment is that the elements of the list that is being built up may themselves be lists. In such a case, flattening the list would flatten the sublists as well. The solution to this problem is shown below.

Use some arbitrary head, rather than `List`, to build up the linked data structure.

```
s = h[];
Do[s = h[s, {i, -i}], {i, 4}]; s
h[h[h[h[h[], {1, -1}], {2, -2}], {3, -3}], {4, -4}]
```

A third argument to `Flatten` specifies that only the head `h` should be flattened. (`Infinity` is a level specification.)

```
Flatten[s, Infinity, h]
h[{1, -1}, {2, -2}, {3, -3}, {4, -4}]
```

Finally, change the outermost head from `h` to `List`.

```
% /. h->List
{{1, -1}, {2, -2}, {3, -3}, {4, -4}}
```

Merging two lists has a very natural recursive implementation, and we'll return to this problem when we discuss methods for speeding up recursion in Section 10.3.

10.2.2 Modifying lists in place

In Section 10.1.7 we pointed out that algorithms that read and write each element of a list have time complexity that is quadratic in the length of the list, rather than linear as might be expected. Writing all results to a second list provides a way around this problem, at the cost of extra memory. Figure 10-2 contains a graphical comparison of the running times of the two moving-average algorithms given in Section 10.1.7, as a function of list size. The upper, quadratic curve represents the algorithm that modifies the list in place; the lower, linear curve represents the algorithm that writes results to a second list.

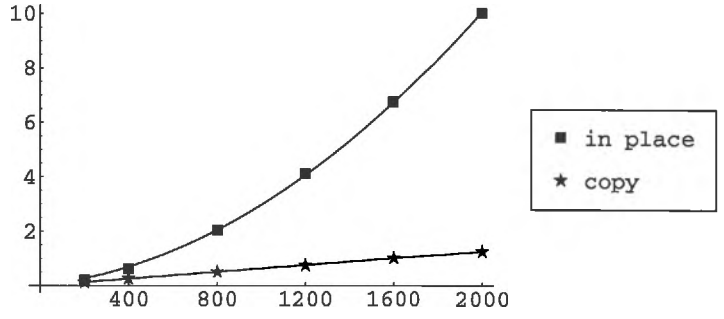


Figure 10-2

For simple computations on lists that are small enough to copy, writing results to a second list works well. However, not all computations are so simple. An excellent example of such a computation is the sorting algorithm known as *quicksort* [Cormen et al. 90]. Quicksort is essentially an extremely sophisticated "shell game" that permutes data into sorted order by swapping strategically chosen pairs of data elements. Quicksort is performed in stages (the actual number of stages depends on the data, although the expected number is close to $\log n$), and the number of data elements that move at each stage typically is less than n. Because of this, quicksort usually moves around many fewer data elements than other sorting algorithms, enabling it to outper-

form them. If all n elements of the data had to be copied to a new list at each stage, quicksort's speed advantage would be negated.

Even for simple computations such as the moving average, it is certainly conceivable that the amount of data being operated upon is so large that there just isn't enough memory to make a copy.

For these reasons, it behooves us to explore other methods for modifying large amounts of data without making copies of the data. We will continue to use the moving-average computation as an example, since it is easy to understand.

Attack 1: Indexed variables

Rather than using a list to hold the data, we can use downvalues. This is a technique that *The Mathematica Book* refers to as *indexed variables*.

Prepare the data.

```
Clear[s]
Do[s[i] = i, {i, 6}]

DownValues[s]
{HoldPattern[s[1]] :> 1, HoldPattern[s[2]] :> 2,
  HoldPattern[s[3]] :> 3, HoldPattern[s[4]] :> 4,
  HoldPattern[s[5]] :> 5, HoldPattern[s[6]] :> 6}
```

The only change to the in-place moving-average algorithm is the replacement of all double square brackets with single ones:

```
Do[s[i] = (s[i] + s[i + 1])/2,
       {i, Length[DownValues[s]] - 1}];

DownValues[s]
{HoldPattern[s[1]] :> 3/2, HoldPattern[s[2]] :> 5/2,
  HoldPattern[s[3]] :> 7/2, HoldPattern[s[4]] :> 9/2,
  HoldPattern[s[5]] :> 11/2, HoldPattern[s[6]] :> 6}
```

Although it's not obvious that using downvalues should be any more efficient than using a list, it turns out to be so, because the kernel uses *hashing* [Cormen et al. 90] to implement near-constant-time lookups of downvalues such as `s[3]`. Figure 10-3 on page 312 shows that the running time of the downvalues technique is linear in the size of the data, although it is not quite as efficient as the list-copying technique.

The use of downvalues rather than lists compromises memory efficiency, because downvalues take up a lot more memory than list elements. We need to find a way to operate directly on lists without taking the performance "hit" associated with them.

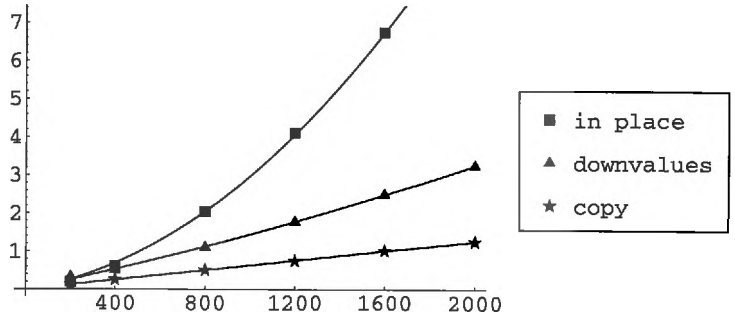


Figure 10-3

Attack 2: HoldComplete

The source of the inefficiency of the in-place list-modifying algorithm is the kernel's insistence on attempting to evaluate the elements of the list each time it is subscripted. We can prevent this overhead by changing the head of the list to `HoldComplete`,[6] as explained in Section 7.3.3. The code for the actual computation doesn't change at all; only the head of the data structure changes.

```
s = HoldComplete @@ Range[1000];
Do[s[[i]] = (s[[i]] + s[[i + 1]])/2,
    {i, Length[s]-1}]; // Timing
{0.633333 Second, Null}

Short[s]

HoldComplete[3/2, 5/2, 7/2, 9/2, <<994>>, 1999/2, 1000]
```

Figure 10-4 shows a graphical comparison of the `HoldComplete`-based moving-average algorithm against the others. It is plain to see that the performance of the `HoldComplete` algorithm is identical to that of the list-copying algorithm, without requiring any extra memory!

From the foregoing discussion it would be easy to draw the conclusion that the `HoldComplete` approach is the automatic choice for procedural operations on large lists. It does have the advantage that it requires only a very small change to the code and yields a qualitative improvement in running time. Nevertheless, the downvalues approach should not be discounted entirely: When the size of the result is not known in advance, it may be the paradigm of choice, since it solves not only the list modification problem but also the `Append` problem discussed in Section 10.2.1.

6. Using `Hold` instead of `HoldComplete` does not achieve the desired effect. See Section 7.3.3 for an explanation of why, as well as a description of an alternative to the `HoldComplete` technique for use with older versions of *Mathematica*.

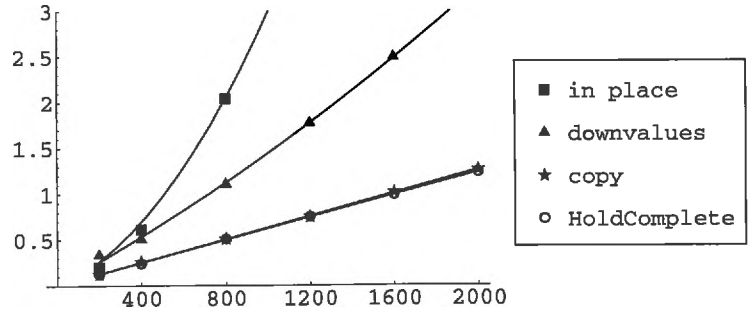


Figure 10-4

Side note: Functional solutions

The techniques discussed in this section are designed for operating on data in place, under the assumption that memory is precious. Freed from this restriction, it is instructive to observe how much faster the moving-average problem can be solved by using functional programming techniques.

Keeping in mind the twin objectives of using built-in functions whenever possible (rule of thumb 1) and operating on as much of the data at one time as possible (rule of thumb 2), we arrive at the following functional solution, which is demonstrated on a list of the first five integers:

Use `Partition` to break up the list into smaller lists that are to be averaged.

```
Partition[Range[5], 2, 1]
{{1, 2}, {2, 3}, {3, 4}, {4, 5}}
```

Sum each of the sublists by applying `Plus` at level 1.

```
Apply[Plus, %, {1}]
{3, 5, 7, 9}
```

Finally, divide by 2.

```
%/2
```

$$\left\{\frac{3}{2}, \frac{5}{2}, \frac{7}{2}, \frac{9}{2}\right\}$$

Figure 10-5 on page 314 shows that the functional solution is much faster than even the fastest linear-time procedural solution.

Incidentally, note how simple it is to generalize this idea to moving averages of arbitrary width (generalizing any of the procedural algorithms in this way requires the introduction of a nested loop, which makes the code even slower):

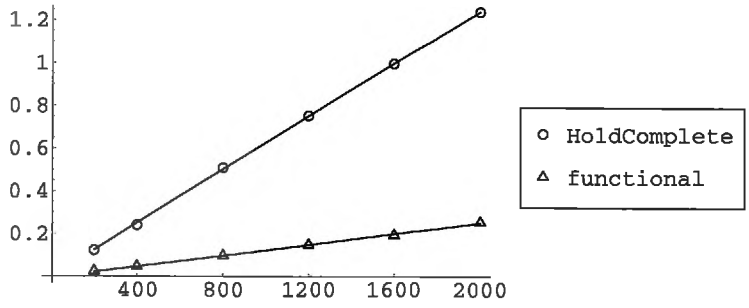


Figure 10-5

```
Table[Apply[Plus,
          Partition[{a, b, c, d, e}, k, 1], {1}]] / k,
      {k, 2, 5}]
```

$$\left\{\left\{\frac{a+b}{2}, \frac{b+c}{2}, \frac{c+d}{2}, \frac{d+e}{2}\right\},\right.$$
$$\left\{\frac{a+b+c}{3}, \frac{b+c+d}{3}, \frac{c+d+e}{3}\right\},$$
$$\left.\left\{\frac{a+b+c+d}{4}, \frac{b+c+d+e}{4}\right\}, \left\{\frac{a+b+c+d+e}{5}\right\}\right\}$$

So what's the catch? Memory efficiency. The functional solution is extravagantly wasteful of memory, because the memory size of the partitioned list is a linear function of the moving-average width k.

```
Table[ByteCount[Partition[Range[1000], k, 1]], {k, 10}]
{44024, 59968, 83856, 99728, 123528, 139328, 163040,
  178768, 202392, 218048}
```

This characteristic precludes the use of the functional technique when both the list size and the width of the moving average are large (as might occur in time-series analysis of large data sets).

Exercise

1. Write a procedural algorithm to perform Gaussian elimination. Compare the performance of your implementation with and without the `HoldComplete` technique.

10.3 Recursion Risks and Rewards

Recursion is a very elegant programming strategy, but there are two problems with using it in *Mathematica:* First, it often doesn't seem to be as fast as other solutions, and second, it can't be applied to very large problems because of evaluation stack size limitations. In this section we'll show that both of these problems can be overcome.

10.3.1 Building up list results

Here is a recursive version of the `merge` function that we coded procedurally in Section 10.2.1.

```
rmerge1[a_List, b_List] :=
    Which[
        a == {} || b == {}, Join[a, b],
        First[a] <= First[b],
            Prepend[rmerge1[Rest[a], b], First[a]],
        True,
            Prepend[rmerge1[a, Rest[b]], First[b]]
    ]

rmerge1[{1, 5, 8}, {2, 3, 4, 7}]
{1, 2, 3, 4, 5, 7, 8}
```

Because deep recursion can crash the kernel, we test this function on small benchmarks, computing averages over several runs to smooth out timing fluctuations. Figure 10-6 contains a graph of the results, along with a quadratic curve fit. Clearly, the time complexity of this algorithm is quadratic.

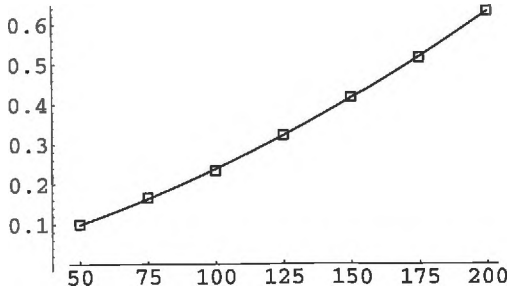


Figure 10-6

The reason for the quadratic complexity of `rmerge1` is the same as in the procedural case: building up lists one element at a time (in this case, using `Prepend`). We can use linked lists to solve that problem here as well. Here is a recursive `merge` function that builds up a linked list of results as the recursion "unwinds."

```
rmerge2[a_List, b_List] :=
    Which[a == {} || b == {}, Join[a, b],
          First[a] <= First[b],
              {First[a], rmerge2[Rest[a], b]},
          True,
              {First[b], rmerge2[a, Rest[b]]}
    ]
```

The result of `rmerge2` is a linked list, as promised.

```
rmerge2[{1, 5, 8}, {2, 3, 4, 7}]
{1, {2, {3, {4, {5, {7, {8}}}}}}}
```

Flatten this structure to get the final answer.

```
Flatten[%]
{1, 2, 3, 4, 5, 7, 8}
```

Figure 10-7 shows a graphical comparison of `rmerge1` and `rmerge2`, with quadratic functions fitted to each. `rmerge2` has a quadratic component, but it is much smaller than before.

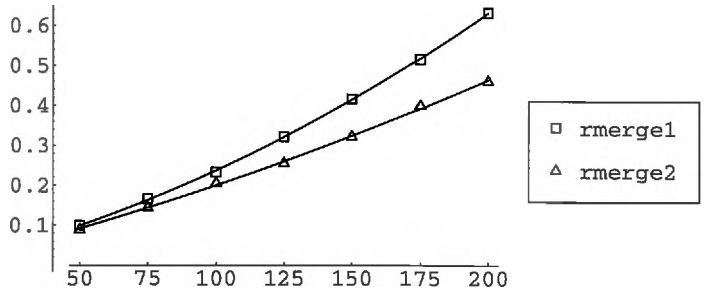


Figure 10-7

Where is that quadratic component coming from? The `Rest` function is the culprit. Every time a recursive call is made, `Rest` copies $n-1$ elements of an n-element list. This is yet another consequence of the fact that *Mathematica* lists are implemented as arrays rather than as linked lists. But this explanation suggests immediately how we can fix the problem: We should use linked-list data structures for the *inputs* to the function as well as for its output. The beauty of this approach is that, assuming the inputs to the recursive `merge` function are in the form of Lisp-style linked lists (e.g., `{e1, {e2, {...{}}}}`, the function needs only a slight modification:

The only change from `rmerge2` is that `Rest[x]` has been replaced by `x[[2]]`.

```
rmerge3[a_List, b_List] :=
    Which[
        a == {} || b == {}, Join[a, b],
        First[a] <= First[b],
            {First[a], rmerge3[a[[2]], b]},
        True,
            {First[b], rmerge3[a, b[[2]]]}
    ]
```

Note that the output and the input have the same structure — very handy!

```
rmerge3[{1,{5,{8,{}}}}, {2,{3,{4,{7,{}}}}}]
{1, {2, {3, {4, {5, {7, {8, {}}}}}}}}
```

In order to use this function conveniently, we need another function that converts a *Mathematica*-style linear list into a linked list. We can write such a function using `Fold` (Section 5.3.3).

```
toLinkedList[s_List] :=
    Fold[{#2, #1} &, {}, Reverse[s]]

toLinkedList[{2, 3, 4, 7}]
{2, {3, {4, {7, {}}}}}
```

Here is a merge from start to finish.

```
Flatten[
    rmerge3[toLinkedList[{1, 5, 8}],
            toLinkedList[{2, 3, 4, 7}]
]   ]
{1, 2, 3, 4, 5, 7, 8}
```

Although the conversion to and from linked lists seems like a lot of extra work, `rmerge3` turns out to be more efficient than `rmerge2` for all but the smallest input sizes (see Figure 10-8). This is because the conversion of the inputs to linked lists is a linear-time operation, as is the flattening of the output; the entire process now consists of a small number of linear-time operations. Hence, superiority is ensured once the problem size is large enough.

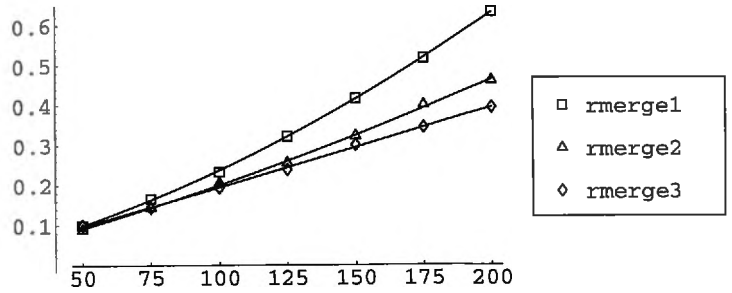


Figure 10-8

10.3.2 Tail recursion

Recursion uses lots of memory for stack space. `$RecursionLimit` can be increased somewhat, but increasing it too much can easily crash the kernel. Is there some way to get the elegance of recursion without running into `$RecursionLimit`?

The answer to this question turns out to be a qualified "yes." Consider the following two functions for computing the length of a list:

This is the straightforward recursive length function.

```
len1[x_] := If[x=={}, 0, 1 + len1[Rest[x]]]
```

```
len1[{a, b, c}]
3
```

This length function is quite tricky.

```
len2[x_] := len2[0, x]
len2[n_, x_] :=
    If[x=={}, n, len2[n + 1, Rest[x]]]


len2[{a, b, c}]
3
```

In `len1` the result of the recursive call is added to `1`. In `len2` the result of the recursive call is returned as the result of the calling function. `len2` works by counting the number of recursive calls *as the calls are made*. When the recursion bottoms out, the result is right there, waiting to be returned.

A function such as `len2` is called *tail-recursive*, which simply means that the recursive call is the last thing the function does before returning. Since the result of the entire recursive computation is ready and waiting as soon as the final recursive call completes, it can be returned directly as the result of the original call! Certain recursive languages, *Mathematica* included, optimize this situation by *not keeping track of tail-recursive calls on the recursion stack*. Not only does this avoid bumping into `$RecursionLimit`, it also speeds up the evaluation (for nontrivial functions)!

Let us verify the above claims.

```
$RecursionLimit = 300;
```

As expected, `len1` cannot deal with a list whose length exceeds `$RecursionLimit`.

```
len1[Table[0, {$RecursionLimit + 1}]]
$RecursionLimit::reclim:
   Recursion depth of 300 exceeded.
298 + If[Hold[{0, 0, 0} == {}], 0,
   1 + len1[Rest[{0, 0, 0}]]]
```

But `len2` has no problem with it.

```
len2[Table[0, {$RecursionLimit + 1}]]
301
```

Timewise, the two functions are very similar.

```
Timing[len1[Table[0, {$RecursionLimit - 10}]]]
{0.216667 Second, 290}

Timing[len2[Table[0, {$RecursionLimit - 10}]]]
{0.216667 Second, 290}
```

Why isn't `len2` faster than `len1`? In theory it should be, but in practice it takes longer to pattern-match each call to `len2`, because there are two rules rather than one and because each call has two arguments rather than one. For a function having a larger amount of computation per recursive call (coming up!) we should see a speed advantage to tail recursion.

This is all very interesting, but is it practical? Clearly the `length` function is a special case; perhaps less-contrived functions cannot be structured so that they are tail-recursive. The `mergesort` algorithm (Section 5.4.2), in which the results of two recursive calls to `mergesort` have to be merged *before* the parent instance of `mergesort` can return, comes immediately to mind. The author knows of no tail-recursive algorithm for `mergesort`. However, necessity often proves to be the mother of invention, and it turns out that the `merge` function can be implemented tail-recursively. This effectively solves the problem with `mergesort`, since `mergesort` itself does not give rise to very deep recursion ($\log n$ recursive calls on an input of length n), whereas `merge` does so (n recursive calls on an input of length n).

Here is a tail-recursive version of the `rmerge3` function (the version that uses linked lists as both inputs and outputs):

```
trmerge1[a_List, b_List] := trmerge1[a, b, {}]
trmerge1[a_List, b_List, c_List] :=
    Which[
        a == {} || b == {}, {c, a, b},
        First[a] <= First[b],
            trmerge1[a[[2]], b, {c, First[a]}],
        True,
            trmerge1[a, b[[2]], {c, First[b]}]
    ]
```

`trmerge1` carries around an extra parameter with it, which is the cumulative result of the merge process. When the recursion bottoms out, the entire merged list (in the form of a linked list) is returned directly. Note that the expression `{c, a, b}` won't be a well-formed linked list; we are taking advantage of the fact that the return value is destined to be flattened. The following example shows the form of the return value from `trmerge1`.

```
trmerge1[toLinkedList[{1, 5, 8}],
            toLinkedList[{2, 3, 4, 7}]
]
{{{{{{{{}, 1}, 2}, 3}, 4}, 5}, 7}, {8, {}}, {}}

Flatten[%]
{1, 2, 3, 4, 5, 7, 8}
```

Figure 10-9 on page 320 clearly shows that the tail-recursive `merge` is a linear-time algorithm, and it works on inputs of arbitrary size without exceeding `$RecursionLimit`. Furthermore, as alluded to earlier, `trmerge1` is slightly faster than its non-tail-recursive counterpart, `rmerge3`, because there is less evaluation stack overhead (see Figure 10-10 on page 320).

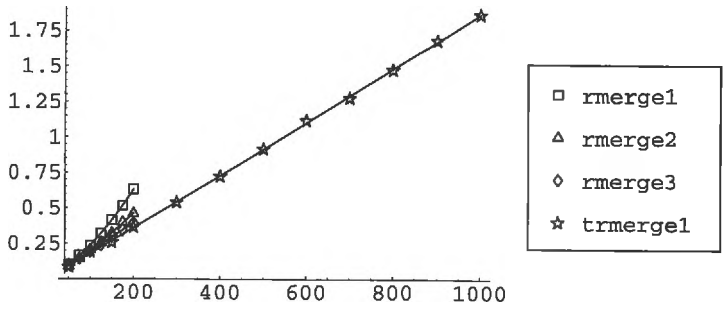


Figure 10-9

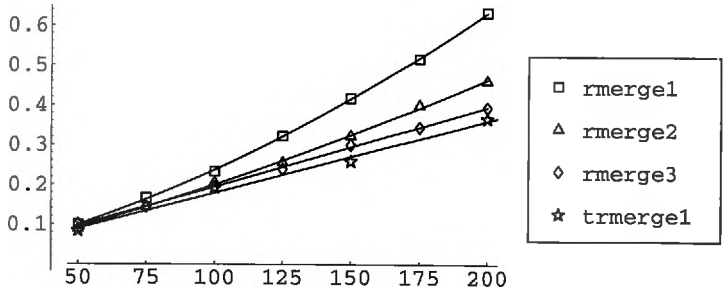


Figure 10-10 Detail view of Figure 10-9.

10.4 Rewrite Rules

We can avoid `$RecursionLimit` by using rewrite rules instead of recursion. In preparation for that, let's convert `rmerge3` into a rule-based recursive function. As before, we'll use the linked-list "tricks" to achieve a linear running time.

```
rbmerge0[a_List, {}] := a
rbmerge0[{}, b_List] := b
rbmerge0[{a1_, arest_}, b:{b1_, _}] :=
    {a1, rbmerge0[arest, b]} /; a1 <= b1
rbmerge0[a:{a1_, _}, {b1_, brest_}] :=
    {b1, rbmerge0[a, brest]}

rbmerge0[toLinkedList[{1, 5, 8}],
         toLinkedList[{2, 3, 4, 7}]] // Flatten
{1, 2, 3, 4, 5, 7, 8}
```

Figure 10-11 shows that `rbmerge0` is faster than `rmerge3`, the function upon which `rbmerge0` is based, and `trmerge1`, the tail-recursive merge function. This is because using the pattern matcher to discriminate between the various cases is faster than using the `Which` function. However, since `rbmerge0` is recursive, it suffers from the same limitations on input size as does `rmerge3`.

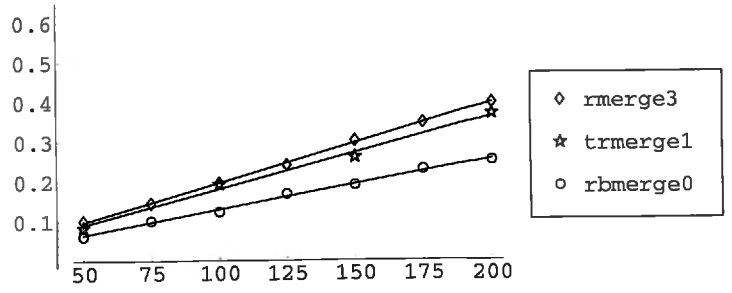


Figure 10-11

Now let's transmogrify `rbmerge0` into a set of stand-alone rules:

We use a new head `mrg` to identify the part of the expression that is still to be merged.

```
mergerules1 = {
    mrg[a_List, {}] :> a,
    mrg[{}, b_List] :> b,

    mrg[{a1_, arest_}, b:{b1_, _}] /; a1 <= b1 :>
        {a1, mrg[arest, b]},
    mrg[a:{a1_, _}, {b1_, brest_}] :>
        {b1, mrg[a, brest]}
};
```

Below is a small example of how these rules operate.

Set up the initial expression.

```
list1 = Take[odds, 5];
list2 = Take[evens, 5];
mrg[toLinkedList[list1], toLinkedList[list2]]
mrg[{1, {3, {5, {7, {9, {}}}}}},
  {2, {4, {6, {8, {10, {}}}}}}]
```

We can watch the rule replacements one at a time.

```
% /. mergerules1
{1, mrg[{3, {5, {7, {9, {}}}}},
   {2, {4, {6, {8, {10, {}}}}}}]

% /. mergerules1
{1, {2, mrg[{3, {5, {7, {9, {}}}}},
     {4, {6, {8, {10, {}}}}}}]}
```

When we're satisfied that it works, we let it run to completion.

```
% //. mergerules1
{1, {2, {3, {4, {5, {6, {7, {8, {9, {10, {}}}}}}}}}}}

Flatten[%]
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
```

The following version of `rbmerge` wraps the special `mrg` head around a pair of linked lists, uses `ReplaceAll` to apply the rules in `mergerules1` until the expression stops changing, and then flattens the result.

```
rbmerge1[a_List, b_List] :=
    Flatten[mrg[toLinkedList[a], toLinkedList[b]] //.
            mergerules1]

rbmerge1[list2, list1]
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
```

Surprisingly, `rbmerge1` has quadratic time complexity (Figure 10-12). It is so slow that it is limited to very small problem sizes.

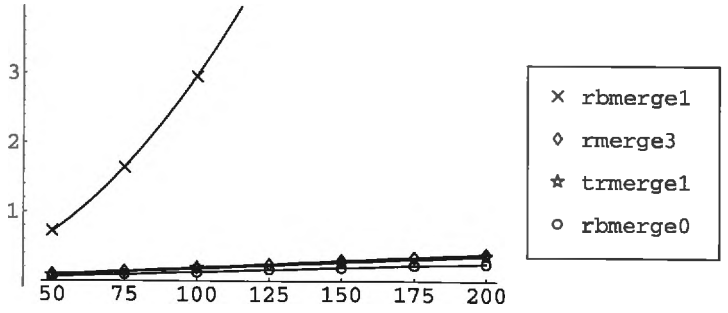


Figure 10-12

Why is `rbmerge1` so excruciatingly slow? Intuitively, we would expect it to be faster than a recursive approach because there is no need to keep track of the evaluation stack. The answer is that the pattern matcher spends a great deal of time trying to match the pattern to parts of the data structure that already have been transformed. To put it another way, the `mrg` head (which is what the pattern matcher is searching for) is getting buried deeper and deeper within the expression as the replacements occur (refer to the example given above). It's not hard to argue that this results in quadratic time complexity.

Is the `ReplaceAll` approach a failure? Not necessarily! If the data structure is set up so that the `mrg` pattern is always the first thing the pattern matcher finds, it will be much faster. The data structure we will use will segregate the "finished" part from the "in progress" part, as shown below:

```
mrg[input1, input2, sortedOutput]
```

All three components of this data structure will be linked lists. The rules now look like this:

```
mergerules2 = {
    mrg[a_List, {}, s_] :> {s, a},
    mrg[{}, b_List, s_] :> {s, b},
    mrg[{a1_, arest_}, b:{b1_, _}, s_] /; a1 <= b1 :>
        mrg[arest, b, {s, a1}],
    mrg[a:{a1_, _}, {b1_, brest_}, s_] :>
        mrg[a, brest, {s, b1}]
};
```

Note the similarity between this set of rules and the tail-recursive function `trmerge1` in the previous section.

Here is an illustration of the operation of the new set of rules. Note the growth of the third list as the other two shrink.

```
mrg[toLinkedList[list1], toLinkedList[list2], {}]
mrg[{1, {3, {5, {7, {9, {}}}}}},
  {2, {4, {6, {8, {10, {}}}}}}, {}]

% /. mergerules2
mrg[{3, {5, {7, {9, {}}}}},
  {2, {4, {6, {8, {10, {}}}}}}, {{}, 1}]

% /. mergerules2
mrg[{3, {5, {7, {9, {}}}}}, {4, {6, {8, {10, {}}}}},
  {{{}, 1}, 2}]
```

The base cases get rid of the empty lists, leaving only the result.

```
% //. mergerules2
{{{{{{{{{{}, 1}, 2}, 3}, 4}, 5}, 6}, 7}, 8}, 9},
  {10, {}}}

Flatten[%]
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
```

The function `rbmerge2` is almost identical to `rbmerge1`. The obvious difference is the use of the new set of rules; a more subtle difference is the extra, empty list in the initial data structure.

```
rbmerge2[a_List, b_List] :=
    Flatten[
        mrg[toLinkedList[a], toLinkedList[b], {}] //.
            mergerules2]

rbmerge2[list2, list1]
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
```

Figure 10-13 compares `rbmerge2` with some of the other linear-time algorithms. Note that it is slightly faster than even `rbmerge0`, the previous champ. Furthermore,

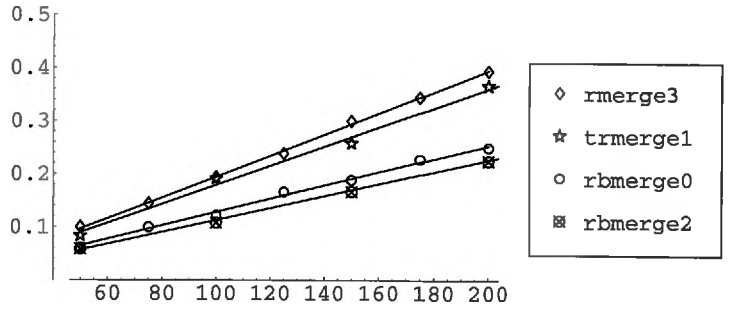


Figure 10-13

`rbmerge2` is immune to `$RecursionLimit` (see Figure 10-14). It is nearly twice as fast as `trmerge1` (the tail-recursive algorithm) over the entire range of tested input sizes, and it's even faster than `pmerge3` (the fastest procedural algorithm) by a comfortable margin.

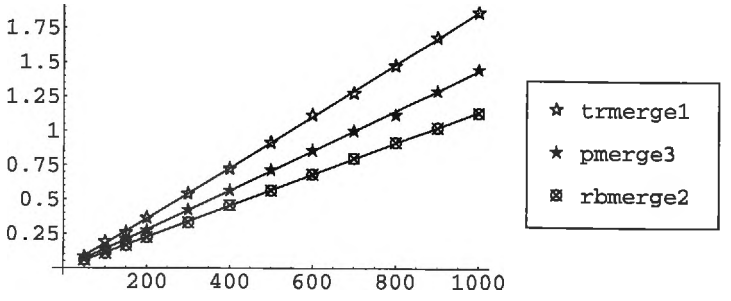


Figure 10-14

Exercise

1. Use the technique developed in this section to speed up the run-length encoding algorithm given in Section 10.1.8. Analyze the running times of the original version and your improved version.

10.5 Compiled Functions

Mathematica has a built-in compilation facility that can be used to speed up certain types of computations. The compiler doesn't translate *Mathematica* code into actual machine-level instructions, because that would not be portable to versions of *Mathematica* running on other machine architectures. Instead, the code is translated into a simple register-based assembly language for a virtual machine that is interpreted by the kernel. Even though this code is interpreted, it still executes much faster than *Mathematica* expressions do because it avoids the main evaluation loop.

10.5.1 An example

Here's a simple example of a compiled function. It takes two arguments and computes the ratio of their logarithms. We also define an uncompiled version of it for comparison.

```
f1[x_, y_] := Log[x]/Log[y]
cf1 = Compile[{x, y}, Log[x]/Log[y]]
```

$$\texttt{CompiledFunction[\{x, y\},}\ \frac{\texttt{Log[x]}}{\texttt{Log[y]}},\ \texttt{-CompiledCode-]}$$

The output of `Compile` is a `CompiledFunction`, which you should think of as being analogous to a `Function` (e.g., pure function). A `CompiledFunction` object contains the parameters to the function, the *Mathematica* version of the code, and a list of pseudocode instructions that print as `-CompiledCode-` in standard output form. You can inspect these instructions using, e.g., `InputForm`, which we'll do shortly.

The compiled function is used in the same way as the uncompiled function. Note, however, that the arguments to a compiled function and all of the arithmetic in the function are assumed to be real.

The integer arguments to `cf1` are evaluated numerically before the compiled function begins executing.

```
{f1[3, 4], cf1[3, 4]}
```

$$\{\frac{\texttt{Log[3]}}{\texttt{Log[4]}},\ 0.792481\}$$

The compiled function is dramatically faster than the uncompiled function.

Here is a list of pairs of real numbers.

```
data = Partition[
        Table[Random[Real, {2., 100.}], {3000}],
        2];
Short[data]
{{90.8874, 5.70024}, <<1498>>, {78.5354, 4.13602}}
```

The compiled function is about 4 times faster than the uncompiled function.

```
Timing[Apply[f1, data, {1}];]
{0.766667 Second, Null}

Timing[Apply[cf1, data, {1}];]
{0.133333 Second, Null}
```

Why is the compiled function so much faster? After all, it is interpreted code. The reason is that the compiler can make a lot of assumptions that the main evaluation loop cannot — e.g., that all inputs and outputs are real, and that all computation can be done using machine arithmetic. This obviates the need to check for such things as variable-precision arguments, upvalues, nonstandard evaluation, and so forth, which really speeds things up.

The parameters to a compiled function are limited in version 2.2 to the types integer, real, complex, or boolean. Version 3.0 adds support for a fifth type, *tensor*, which we will discuss in Section 10.5.5. The default type is real; we'll see how to specify the other parameter types in Section 10.5.3.

Compiled evaluation will fail if the actual argument types are incorrect and cannot be coerced into the correct type (i.e., integer to real as in the above example). When this happens, the compiled function uses the original *Mathematica* code to perform the evaluation.

```
cf1[True, 1 + I]
CompiledFunction::cfr:
   Cannot use compiled code; Argument True at
      position 1 should be a machine-size real number.
Log[True]
-----------
Log[1 + I]
```

This sort of behavior should be avoided if possible, since this is slower than evaluating the *Mathematica* code in the first place. Another, more subtle limitation is that the results of all computations are assumed to be real; the compiled evaluation will fail if they are not.

```
cf1[-2, -3]
CompiledFunction::cfn:
   Numerical error encountered at instruction 4;
     proceeding with uncompiled evaluation.
I Pi + Log[2]
-------------
I Pi + Log[3]
```

Both of these problems will be dealt with shortly; but first, let's take a look at the internal form of a compiled function.

10.5.2 Compiled code

We can use `InputForm` to inspect the actual pseudocode contained in a `CompiledFunction`.

```
InputForm[cf1]
CompiledFunction[{_Real, _Real}, {0, 0, 6, 0, 0},
  {{1, 2}, {4, 1, 0}, {4, 2, 1}, {54, 0, 2},
   {54, 1, 3}, {45, 3, 4}, {38, 2, 4, 5}, {9, 5}},
  Function[{x, y}, Log[x]/Log[y]]]
```

`InputForm` reveals that the internal form of a `CompiledFunction` is quite different from its `OutputForm`. There actually are four parts to a `CompiledFunction`. The first part is a list of the types of the arguments. (The arguments are unnamed because they are referred to by number within the compiled code.) The second part is a list of the number of registers of each type that the code uses, in the format {*nboolean*, *nint*, *nreal*, *ncomplex*, *ntensor*} (the final element in this list is absent in versions prior to 3.0). This particular compiled function uses six real registers. The third part is a list of pseudocode instructions, and the final part is a pure function that is used in case the compiled evaluation fails.

Let's go through the pseudocode instructions[7] for `cf1`. The first instruction, `{1, 2}`, asserts that the version of the instruction set that was used to generate this code is version 2. The next two instructions load the parameters into registers; each instruction is of the form {4, *param#*, *realreg#*}. The next two instructions are logarithms: {54, *r1*, *r2*} takes the `Log` of the contents of real register *r1* and places the result in real register *r2*. (This is the instruction that failed in the preceding example.) Opcode 45 is a real reciprocal, hence {45, 3, 4} places `1/Log[`*param1*`]` into real register 4. Opcode 38 is real multiplication; the product of real registers 2 and 4 is placed in real register 5. Finally, opcode 9 returns the contents of a real register, in this case register 5.

10.5.3 Specifying types

The general form of the parameter list to `Compile` is {{*name1*, *type1*}, {*name2*, *type2*}, ...}. The types are specified as `_Integer`, `_Real`, `_Complex`, and `True | False` for boolean. (We defer discussion of the tensor type to Section 10.5.5.) Note that types are not specified the way they are for ordinary function declarations, e.g., `c_Complex`, and that each {*name*, *type*} pair must be in its own sublist, even if there is only a single parameter.

For example, here is a function that takes a boolean and a complex number as arguments; if the boolean is `True`, the function returns the conjugate of the complex number, otherwise it returns the original number.

7. Note that some of these instructions will be different in versions prior to 3.0. A listing of all pseudocode instructions for version 2.2 can be found in [Keiper 93a]. At the time of this writing there is no official source for the pseudocodes in version 3.0, although one will surely appear on *MathSource* before long.

```
cf2 = Compile[{{b, True | False}, {c, _Complex}},
        If[b, Conjugate[c], c]]
CompiledFunction[{b, c}, If[b, Conjugate[c], c],
  -CompiledCode-]

cf2[#, 3 + 4I]& /@ {True, False}
{3. - 4. I, 3. + 4. I}
```

As mentioned earlier, in the absence of type declarations the compiler assumes that arguments are real and that the results of all computations will be real as well. However, if an argument is declared as complex, then the compiler will assume that all functions of that argument are complex. Here's an example.

The parameter `x` is assumed to be of type `Real`.

```
InputForm[cf3 = Compile[{x}, Log[x]]]
CompiledFunction[{_Real}, {0, 0, 2, 0, 0},
  {{1, 2}, {4, 1, 0}, {54, 0, 1}, {9, 1}},
  Function[{x}, Log[x]]]
```

The parameter `x` is explicitly declared to be of type `Complex`.

```
InputForm[cf4 = Compile[{{x, _Complex}}, Log[x]]]
CompiledFunction[{_Complex}, {0, 0, 0, 2, 0},
  {{1, 2}, {5, 1, 0}, {68, 0, 1}, {10, 1}},
  Function[{x}, Log[x]]]
```

Aside from the fact that `cf3` uses real registers and `cf4` uses complex registers, note that the opcode used to compute the logarithm is different in the two functions. Opcode `54` computes the log of a real register and puts the result in a real register, whereas opcode `68` computes the log of a complex register and puts the result in a complex register. Thus, `cf4` will work even for negative arguments.

```
cf3[-2]
CompiledFunction::cfn:
   Numerical error encountered at instruction 3;
     proceeding with uncompiled evaluation.
I Pi + Log[2]

cf4[-2]
0.693147 + 3.14159 I
```

10.5.4 Uncompilable expressions

Now that we have discussed argument types, it's time to explain what the compiler can and cannot do. The compiler's repertoire is limited to arithmetic and logical operations, elementary functions, and procedural constructs such as `If`, `Do`, `Module`, etc. (The version 3.0 compiler also can compile most list operations; see Section 10.5.5.) Whenever the expression to be compiled contains any unknown functions — and this includes many built-in functions as well as all user-defined functions — the compiler generates instructions that evaluate those functions using the standard evaluation process. Here's an example:

There is no pseudocode instruction for the `LogIntegral` function.

```
f5[z_] := z + LogIntegral[z]
InputForm[cf5 = Compile[{z}, z + LogIntegral[z]]]
CompiledFunction[{_Real}, {0, 0, 3, 0, 0},
  {{1, 2}, {4, 1, 0},
   {31, Function[{z}, LogIntegral[z]], 3, 0, 1},
   {34, 0, 1, 2}, {9, 2}},
  Function[{z}, z + LogIntegral[z]]]
```

The instruction `{31, expr, type, rank, reg}` evaluates *expr* using the normal evaluation process. The *type* element encodes the expected type of the result as an integer from 1 to 4, corresponding to logical, integer, real, and complex types.[8] The *rank* element is used for tensor operations. In the example function, since there is no opcode for the `LogIntegral` function, an opcode `31` instruction is constructed for the expression `Function[{z}, LogIntegral[z]]`. This will slow down the execution of the compiled function a great deal:

```
newdata = Range[2., 1000.];
Short[newdata]
{2., 3., 4., 5., 6., <<991>>, 998., 999., 1000.}
```

The uncompiled function takes this long.

```
Timing[f5 /@ newdata;]
{6.48333 Second, Null}
```

The compiled evaluation is slower!

```
Timing[cf5 /@ newdata;]
{6.75 Second, Null}
```

Furthermore, since the argument to `LogIntegral` is real, the compiler expects the result to be real as well. This can cause problems, because `LogIntegral` is not a real-valued function for negative real arguments:

```
cf5[-2.5]
CompiledFunction::ccr:
   Expression -0.00285625 + 3.88437 I
     should be a machine-size real number.
CompiledFunction::cfex:
   External evaluation error at instruction 3;
     proceeding with uncompiled evaluation.
-2.50286 + 3.88437 I
```

One way around this problem is to declare the parameter `z` as complex, but that would cause all of the compiled operations on `z` to be done with complex registers, which could take about twice as long as with real registers (the compiler uses C doubles for real numbers internally, and a pair of C doubles for complex numbers). As an alternative, you can tell the compiler to expect certain functions to return certain types using

8. In versions prior to 3.0, there is a different opcode for each possible type of the result; e.g., opcode 24 evaluates an expression to get a real result, opcode 25 does the same thing for complex results, etc.

an optional third argument to `Compile` of the form `{{pattern1, type1}, {pattern2, type2}, ...}`:

The third argument tells `Compile` to expect `LogIntegral` to return a complex value.

```
cf6 = Compile[{z}, z + LogIntegral[z],
                {{LogIntegral[_], _Complex}}]
CompiledFunction[{z}, z + LogIntegral[z],
  -CompiledCode-]
```

Now there is no error.

```
cf6[-2.5]
-2.50286 + 3.88437 I
```

Examining the pseudocode for `cf6` reveals that it stores the result of evaluating `LogIntegral[z]` in a complex (type 4) register rather than a real one. It then coerces `z` (which is in real register 0) into a complex number by storing `z + 0. I` in complex register 2, and the arithmetic from that point on is complex.

```
InputForm[cf6]
CompiledFunction[{_Real}, {0, 0, 2, 3, 0},
  {{1, 2}, {4, 1, 0},
   {31, Function[{z}, LogIntegral[z]], 4, 0, 0},
   {15, 0., 1}, {25, 0, 1, 1}, {35, 1, 0, 2},
   {10, 2}}, Function[{z}, z + LogIntegral[z]]]
```

Uncompilable expressions inside of a compiled function can render compilation useless or even counterproductive, as we saw above. There are some particularly treacherous cases for which it's not obvious that compilation won't work. One such case is the use of global variables. Every global variable in a compiled expression will go through standard evaluation each time it is used; this can virtually cripple the compiled function if the variable is something like a loop index. Consider the following examples:

```
f7[lim_] := (s = 0; Do[s += i, {i, 1, lim}]; s)
cf7 = Compile[{{lim, _Integer}},
            s = 0; Do[s += i, {i, 1, lim}]; s];

Timing[f7[5000]]
{0.766667 Second, 12502500}
```

The compiled function is nearly 8 times slower!

```
Timing[cf7[5000]]

{6.05 Second, 1.25025 10^7}
```

Here's why: A pure function is being evaluated for every expression involving either `s` or `i`.

```
Cases[cf7, Function[___], Infinity]
{Function[{lim}, s], Function[{lim}, s = 0],
  Function[{lim}, s], Function[{lim}, s = s + i],
  Function[{lim}, s],
  Function[{lim}, s = 0; Do[s += i, {i, 1, 5000}]; s]}
```

You can get around this problem by making sure that symbols such as `s` and `i` are declared locally in the body of the compiled function.

```
cf8 = Compile[{{lim, _Integer}},
                Module[{i, s = 0},
                    Do[s += i, {i, 1, lim}]; s
                ]
            ]
CompiledFunction[{lim},
  Module[{i, s = 0}, Do[s += i, {i, 1, lim}]; s],
  -CompiledCode-]
```

Quite a difference!

```
Timing[cf8[5000]]
{0.0833333 Second, 12502500}
```

Incidentally, the compiler *infers* the type of a local variable from the first assignment to that variable that it comes across. Thus you may sometimes see strange warning messages such as the following:

```
cf9 = Compile[{{lim, _Integer}},
                Module[{i, s = 0},
                    Do[s += Sqrt[i], {i, 1, lim}]; s
                ]
            ]
Compile::cset:
   Warning: Variable s of type _Integer
     encountered in assignment of type _Real.
CompiledFunction[{lim},
  Module[{i, s = 0}, Do[s += Sqrt[i], {i, 1, lim}];
    s], -CompiledCode-]
```

The problem here is that `s` was initially assigned the integer `0`, so the compiler assumed that `s` was an integer and loaded it into an integer register. Subsequently, the compiler is unable to figure out how to compile `s += Sqrt[i]` since `s` is an integer, but the compiler thinks that `Sqrt` should return a real. If you try to execute this function, the compiled evaluation will fail, resulting in a very inefficient standard evaluation:

```
cf9[10]
CompiledFunction::ccint:
   Expression 1 + Sqrt[2]
     should be a machine-size integer.
CompiledFunction::cfex:
   External evaluation error at instruction 31;
     proceeding with uncompiled evaluation.
6 + 3 Sqrt[2] + Sqrt[3] + Sqrt[5] + Sqrt[6] +
  Sqrt[7] + Sqrt[10]
```

The way to get around this problem is to make sure that the initial assignment to `s` is a real number, as shown below. Then the compiler will infer that `s` is a real variable.

```
cf10 = Compile[{{lim, _Integer}},
          Module[{i, s = 0.},
              Do[s += Sqrt[i], {i, 1, lim}]; s
          ]
      ];

Timing[cf10[5000]]
{0.0833333 Second, 235737.}
```

The resulting compiled function is an order of magnitude faster than even a fast functional implementation of this computation (Section 10.1.3):

```
Plus @@ Sqrt[Range[1., 5000., 1.]] // Timing
{0.933333 Second, 235737.}
```

10.5.5 The tensor type

In version 3.0 the compiler supports an additional type, the *tensor*. A tensor is a generalized vector; in *Mathematica*, a tensor is a nested list in which every element at the same level has the same number of subelements. The number of levels in a tensor is called its *rank*. For example, a list is a rank-1 tensor, and a matrix is a rank-2 tensor.

In order to be used by the compiler, all elements of a tensor must be one of the three numeric types supported by the compiler (integer, real, or complex), and all of the elements within a given tensor must be of the same type. Furthermore, the compiler needs to know the rank of a tensor at compile time in order to generate code for operations on it. Thus, tensor parameters have one extra element in their declaration, namely, the rank of the tensor. For example, the following compiled function sums the elements of a list that is passed as a parameter:

`s` is a rank-1 tensor with real elements.

```
csum1 = Compile[{{s, _Real, 1}},
    Module[{i = 0, sum = 0.},
        For[i = 1, i <= Length[s], i++, sum += s[[i]]];
        sum
    ]
];
```

Sum a list of 100,000 real numbers.

```
z = Range[1., 100000];
Timing[csum1[z]]

{1.36667 Second, 5.00005 10^9}
```

`csum1` is much faster than an uncompiled `For` loop.

```
sum = 0;
Timing[For[i = 1, i <= Length[z], i++, sum += z[[i]]]]
{50.8667 Second, Null}
```

It is even faster than the built-in `Sum` function.

```
Timing[Sum[z[[i]], {i, Length[z]}]]

{9.83333 Second, 5.00005 10^9}
```

However, a functional solution is still faster.

```
Timing[Plus @@ z]
{0.983333 Second, 5.00005 10^9}
```

The compiler can generate code for virtually all of the built-in list operations, such as `Append`, `Prepend`, `Drop`, `Take`, `Sort`, `Partition`, `RotateLeft`, `Join`, and so forth. It can even generate code for higher-level functional operations such as `Map`, `Apply`, `Nest`, and `Fold`. Here is a compiled version of the last computation demonstrated above.

```
csum2 = Compile[{{s, _Real, 1}}, Plus @@ s];
```

It is even faster than the built-in `Apply`.

```
Timing[csum2[z]]
{0.283333 Second, 5.00005 10^9}
```

Since the code generated by the compiler for tensor operands depends on the rank of a tensor, it is an error to change the rank of a tensor during the execution of a compiled function; attempting to do so will cause the execution to proceed with the uncompiled code. Thus, for example, it is not possible to use the compiler to operate on linked lists of the type introduced in Section 10.3.1.

As is the case for the other types, in order to use a local variable as a tensor, the first assignment to that variable must be a tensor of the appropriate type and rank. For example, here is a compiled function for computing the moving average of a list of real numbers. (The second argument gives the width of the average.) Note the form of the initial assignment to the local variable `t`:

```
cma = Compile[{{s, _Real, 1}, {k, _Integer}},
    Module[{t = Table[0., {Length[s] - k + 1}],
            n = Length[s] - k + 1},
        Do[t[[i]] = Sum[s[[i + j]], {j, 0, k - 1}]/k,
           {i, n}];
        t
    ]
];
```

Compiled code using tensor operations is not always faster than the corresponding uncompiled code. Since the internal representation used by the compiler for tensors is different than the representation used by the kernel for nested lists, there is some overhead involved in converting from one to the other. Furthermore, the internal representation used by the compiler is inherently less efficient for tensors of high rank (i.e., deeply nested lists). Trial and error seems to be the only way to find out which approach is more efficient.

Although the author has not yet discovered any hard-and-fast rules about the new tensor type, in general it seems that computations involving lots of numerical operations are sped up a great deal by compilation, whereas computations that involve lots of data movement but very little computation (such as merging two lists) do not benefit very much.

10.6 Additional Resources

Two recent articles in the *Mathematica Journal* ([Gayley 94a], [Hayes 95]) deal with a number of techniques for writing efficient programs.

A good description of internal compilation can be found in [Keiper 93a]. A list of opcodes for version 2.2 of the compiler can be found on *MathSource* in item #0201-889, "The *Mathematica* Compiler."

11

MathLink

MathLink is a cross-platform communication protocol defined by Wolfram Research, Inc. There is a C-library interface to *MathLink* as well as a *Mathematica* command-based interface. *MathLink* typically is used to communicate between a *Mathematica* kernel and a program written in a compiled language such as C, although it can also be used for kernel-to-kernel communication or even C program-to-C program communication (without any *Mathematica* involvement whatsoever).

There are two typical uses for *MathLink*. The first of these is to allow the kernel to call functions written in another programming language. These functions may be part of a pre-existing scientific application library, or they may have been written specifically for the purpose of performing some computation that is inefficient in *Mathematica*.

The other common use for *MathLink* is to allow a C program to exploit the computational power of the *Mathematica* kernel, or to provide a nicer interface to it. In fact, the front end communicates with the kernel using *MathLink*, which should give you some appreciation of the protocol's versatility. An example of an alternative front end is the *MathLink for Excel and Mathematica* program that is distributed by WRI, which allows users to enter arbitrary *Mathematica* expressions into spreadsheet cells.

In the present context of performance enhancement, we are concerned mainly with the first use of *MathLink*, that is, calling other programs from *Mathematica*. We will, however, discuss how such programs can send *Mathematica* expressions back to the kernel for evaluation, which lays the foundation for constructing programs of the second type. We also will discuss using *MathLink* for kernel-to-kernel communication, which can be used to implement large-grained parallel processing.

At the time of this writing the only compiled language supported directly by *MathLink* is C. However, this does not exclude the use of other languages (e.g., Fortran),

since on most systems it is possible to link C code with code written in other languages, and the amount of code that absolutely *has* to be written in C is very minimal.

11.1 *MathLink* Fundamentals

In this section we will discuss the most basic aspects of the *MathLink* C language interface. Although it may seem as though there is a lot of drudgery involved, fear not! There are some excellent higher-level tools provided with the *MathLink* distribution that simplify the construction of *MathLink* programs enormously. We'll begin introducing these tools in the next section. However, as with all things involving computers and especially programming, some knowledge of the fundamentals can go a long way toward figuring out what to do when the "canned" solutions don't work.

11.1.1 Hello world

We begin with the *de rigueur* demonstration of any programming language or paradigm: the "hello world" program. Do not be dismayed by the apparent complexity of the code shown below — over 90 percent of it (everything but the call to `MLPutString`) is "boilerplate" initialization and cleanup code that can be used in any *MathLink* program.[1]

This header file must be included in all *MathLink* programs.

These data types are defined in `mathlink.h`.

Initialize the *MathLink* runtime environment and open a link.[2]

Send a string to the other side.

Wait until the other side of the link is closed.

Close the link and clean up before exiting.

```
#include "mathlink.h"

main(int argc, char **argv)
{   MLEnvironment mlenv;
    MLINK mlink;

    mlenv = MLInitialize((MLParametersPointer)0);
    mlink = MLOpenArgv(argc, argv);

    MLPutString(mlink, "Hello world!");

    MLGetNext(mlink);

    MLClose(mlink);
    MLDeinitialize(mlenv);
    return 0;
}
```

1. N.B.: If you are using the MacOS operating system, your program must initialize the Macintosh toolbox before using any *MathLink* calls, or else the program may crash. See the documentation that comes with the *MathLink* distribution for MacOS if you don't know how to do this.
2. Prior to version 3.0, `MLOpenArgv` was called `MLOpen`.

After this code is compiled, it must be linked together with the *MathLink* library; see the documentation that comes with the *MathLink* distribution for your computer for the details. Let us assume that we have successfully compiled and linked this program into an executable file called *hello*. Here is how the *hello* program would be called from *Mathematica*:

```
link = LinkLaunch["hello"]
LinkObject[hello, 2, 2]

LinkRead[link]
Hello world!

LinkClose[link]
```

The `LinkLaunch`[3] command launches and opens a link to the *hello* program, which for the purposes of this example is located in the same directory as the *Mathematica* kernel. (If the file resided elsewhere, we would have had to specify its full pathname.) `LinkRead` reads a data item from the link, which in this case is the string sent by *hello*. Finally, `LinkClose` closes the link; at that point, the `MLGetNext` call in *hello* returns an error status, which causes that program to clean up things on its end and terminate.

11.1.2 Two-way communication

Our next example is a function that sets a given bit in an integer. This is a task for which *Mathematica* is not well suited; it would be necessary to break the integer up into binary digits using `IntegerDigits[num, 2]`, manipulate the resulting list of `0`s and `1`s, and then reconstitute the answer as another integer using Horner's rule (Section 5.3.3, "Fold"). Although this is certainly doable, it's ugly and inefficient. On the other hand, this kind of "bit bashing" is easy to do in a low-level language like C.

The external program shown below expects the kernel to send it two arguments, the integer and a bit number, and the program returns the result of setting the given bit number in the given integer. The boilerplate from the *hello* example can be used unchanged; merely declare the following variables at the beginning of `main`:

```
int n;
int b;
```

and replace the call to `MLPutString` with the three calls shown below:

`&name` passes *name* as a by-reference parameter.

`<<` is left-shift and `|` is logical OR.

```
MLGetInteger(mlink, &n);
MLGetInteger(mlink, &b);
MLPutInteger(mlink, n | 1<<b);
```

3. `LinkLaunch` is new to version 3.0. Users of earlier versions should use `LinkOpen` instead.

As you can see, the input counterpart to `MLPuttype` is `MLGettype`. There are `MLPuttype` and `MLGettype` functions for nearly all of the basic C types. Note that the second argument to each `MLGettype` call is passed by reference.

Here's how this external program is used from *Mathematica*:

```
link = LinkLaunch["setbit"];

LinkWrite[link, 1];
LinkWrite[link, 5];
LinkRead[link]
33

LinkClose[link];
```

Set bit 5 in the integer 000001_2.

There are a few points worth noting about this example. First of all, in *Mathematica* only two functions — `LinkRead` and `LinkWrite` — suffice for reading and writing data of any type to or from a link. On the other hand, in the C interface there is a different function for putting and getting each basic C type. This is required since C is a *statically typed* language, that is, the compiler needs to know the type of every parameter that is passed to a function. A consequence of this is that it's possible for a *MathLink* program to try to get a type of data from the link that doesn't match the datum that's actually waiting on the link! When this happens, an error condition is raised on the link which, if not dealt with, causes subsequent input operations from that link to return without reading anything.

For example, if the first expression written to the link by the kernel had been a symbol, string, floating-point number, extended-precision integer, or any nonatomic expression, both `MLGettype` calls would have failed, and the program would have written garbage to the link. Error handling is one of the trickiest aspects of writing *MathLink* programs. Fortunately, the tools that we will discuss in Section 11.2 write most of this code for you.

Exercises

1. Write a *MathLink* program that takes two integers n and b, where $0 \le b \le 31$, and returns bit b of n.
2. Write a *MathLink* program that takes a string and two integers n_1 and n_2 as arguments and returns the substring that begins at position n_1 and ends at position n_2. (You will need to use the *MathLink* library function `MLGetString`.)

11.1.3 A *MathLink* server program

The program of the preceding section, in addition to not being very robust, is rather inefficient and user-unfriendly. First of all, it would be nice if the program would continue to execute after answering a request from the kernel, so that it could be launched once but used repeatedly. This is quite easy to fix:

```
while(MLGetInteger(mlink, &n) &&
      MLGetInteger(mlink, &b))
   MLPutInteger(mlink, n | 1<<b);
```

The `While` loop checks the return values of the `MLGettype` calls, which are either nonzero on success or zero on failure. In this way, the program will continue to execute for as long as the kernel keeps the link open, barring any errors, of course. When the kernel closes the `link`, the `MLGettype` calls will fail, causing the program to exit from the `While` loop and terminate. (Note that the call to `MLGetNext`, from the original boilerplate, is no longer necessary to keep the program from terminating prematurely.) Thus we have constructed a simple *server* program.

On the kernel side, the server program can be made easier to use by hiding the *MathLink* details inside of a function definition like the following:

This definition assumes that an active link to the server program is contained in `link`.

```
SetBit[n_, b_] :=
(   LinkWrite[link, n];
    LinkWrite[link, b];
    LinkRead[link]
)
```

Furthermore, there's no reason why the same server program can't provide more than one service, as long as the kernel prefaces the data it sends with some indication of the service it is requesting. Here is a C code fragment that provides two services to the kernel, setting a bit and clearing a bit:

```
char *func;

while (MLGetString(mlink, &func) &&
       MLGetInteger(mlink, &n) &&
       MLGetInteger(mlink, &b))
{   if (!strcmp(func, "setbit"))
        MLPutInteger(mlink, setbit(n, b));
    else if (!strcmp(func, "clearbit"))
        MLPutInteger(mlink, clearbit(n, b));
    else
        MLPutSymbol(mlink, "$Failed");

    MLDisownString(mlink, func);
}
```

The new argument, `func`, is the name of the service being requested.

`MLDisownString` is a memory management operation; see Section 11.4.3.

This program expects three arguments from the kernel: a string called `func`, indicating the name of the service that is being requested, and the two arguments from the previous example. (Naturally, if different services required different arguments, then the program's logic would be slightly more complex.) Note that if `func` is not one of the strings `"setbit"` or `"clearbit"`, the program returns the *symbol* `$Failed`.[4]

4. The `Symbol` type, like `String`, is represented in C as a character array. The difference lies in how the characters are interpreted by the kernel.

The code that actually sets and clears bits has been separated from the main routine to emphasize the distinction between communication and computation. Here are the definitions of `setbit` and `clearbit`:

```
int setbit(int n, int b)
{   return n | 1<<b; }

int clearbit(int n, int b)
{   return n & ~(1<<b); }
```

Here is the new *Mathematica* definition for `SetBit`;[5] the definition for `ClearBit` is analogous.

```
SetBit[n_, b_] :=
(   LinkWrite[link, "setbit"];
    LinkWrite[link, n];
    LinkWrite[link, b];
    LinkRead[link]
)
```

Below we demonstrate the features of this program.

```
link = LinkLaunch["setbit"];
```

The server can be invoked multiple times on the same launch.

```
SetBit[1, 5]
33

ClearBit[%, 0]
32
```

If the name of the function is not `"setbit"` or `"clear-bit"`, the symbol `$Failed` is returned.

```
LinkWrite[link, "unknown function"];
LinkWrite[link, 0];
LinkWrite[link, 0];
LinkRead[link]
$Failed
```

The program continues to operate, however.

```
SetBit[3, 7]
131
```

When the link is closed, the server program terminates.

```
LinkClose[link];
```

This is about as far as we will go into writing *MathLink* programs from scratch. The next section introduces some tools that can generate most or all of the communication code for a *MathLink* server program from a high-level description of the services that the program provides. However, from time to time we will need to use *MathLink* calls

5. To avoid confusion, we use lowercase names for C functions and capitalized names for the corresponding *Mathematica* functions.

like the ones demonstrated in this section to deal with situations that are outside the purview of the tools.

Exercises

1. Add a function called `getbit` to the server program that gets the value of a particular bit in an integer.
2. Write a *Mathematica* function called `SubString` that provides a nicer interface to the substring extraction *MathLink* function that you created in Exercise 11.1.2.2.
3. Modify the substring extraction program so that it remains active in between calls to `SubString`.

11.2 Template-Based *MathLink* Programs

In the last section we developed, from the ground up, a simple *MathLink* server program and a *Mathematica* interface corresponding to it. This is such a common use of *MathLink* that some tools exist for automating a large part of this process. The basic idea behind these tools is that you provide C source code for functions that you would like to be able to call from *Mathematica*, and a high-level description — a *template* — of the interface to these functions. The tools then generate all of the communication-related code for you and combine it with your function definitions to produce a complete server program. The C functions that have been encapsulated in this way are called *installable* functions for reasons that will become clear later.

The tools that accomplish this feat are called *mprep* and *mcc*.[6] *mprep* reads a file called a *template file* that specifies, for each installable function, such things as the name of the function, the types of parameters it expects, and the name that the kernel should use to refer to the function. *mprep* translates this information into a set of C language functions that handles all of the communication details for the *MathLink* program and makes ordinary C language function calls to the installable functions. Then an appropriate C compiler or integrated development environment is used to compile and link the *mprep*-generated code with the file(s) containing the installable functions and the *MathLink* library. In the simplest cases no changes to the functions' code are required, and a very minimal amount of interface code needs to be written.

6. *mprep* is the name of a program that is invoked from the command line on UNIX and DOS/Windows systems, and *mcc* is a UNIX shell script that invokes *mprep*. The *MathLink* distribution for MacOS includes both an MPW-shell version of *mprep* and a double-clickable application file called *SAmprep* (for "stand-alone *mprep*").

11.2.1 Making setbit and clearbit installable

For our first example of a template-based program, we will turn the C functions `setbit` and `clearbit` from the last section into installable functions. Here is *all* of the C code that we need to write:

```
#include "mathlink.h"

int main(int argc, char **argv)
{   return MLMain(argc, argv);
}

int setbit(int n, int b)
{   return n | (1<<b);
}

int clearbit(int n, int b)
{   return n & ~(1<<b);
}
```

A single line of "glue code" has to be written to interface with the *mprep*-generated code.

The functions being installed require no modifications in this example.

It is the programmer's responsibility to write the `main` routine, but the only requirement for `main` is that it call the function `MLMain` (which is written by *mprep*), passing to it the command-line arguments that it receives from the operating system.[7] `MLMain` opens the link and then enters a loop, translating *MathLink* messages from the kernel into ordinary function calls to `setbit` and `clearbit`, and translating their return values into *MathLink* replies. Finally, when the kernel closes the link, `MLMain` returns and the programmer's `main` function resumes control. By structuring the control flow in this way the programmer has an opportunity to perform whatever initialization is necessary before relinquishing control to `MLMain`, and an opportunity to clean up after `MLMain` returns.

The only other code that we must write is a set of *template* descriptions for the functions being installed. The templates for `setbit` and `clearbit` are shown below:

```
:Begin:
:Function:      setbit
:Pattern:       SetBit[n_, b_]
:Arguments:     {n, b}
:ArgumentTypes:{Integer, Integer}
:ReturnType:    Integer
:End:
```

7. Command-line arguments are another UNIX/DOS-specific feature. If you are programming under MacOS, your development environment should have a compatibility library that allows a program to receive these arguments. The library typically puts up a dialog box when the program begins running that prompts the user for the arguments.

```
:Begin:
:Function:      clearbit
:Pattern:       ClearBit[n_, b_]
:Arguments:     {n, b}
:ArgumentTypes:{Integer, Integer}
:ReturnType:    Integer
:End:
```

Each template is delimited by `:Begin:` and `:End:` statements. The `:Function:` statement specifies the C language name of the installable function. The `:Pattern:` statement gives a pattern that the kernel will use to create a *Mathematica* definition for the installable function. The `:Arguments:` statement tells the kernel what expressions to send to the external program when the corresponding function is called. (In this simple case, the arguments are just the pattern variables in the `:Pattern:` statements, but in general they can be any *Mathematica* expressions.)

The `:ArgumentTypes:` and `:ReturnType:` statements are necessary because, as noted earlier, there is a different `MLGettype` (or `MLPuttype`) call for each of the basic C types and *mprep* needs to know which ones to use. This information also is needed in order to generate C function calls that will link correctly to the `setbit` and `clearbit` functions. Note that the type specifications are special keywords, not C language types; there is a predefined correspondence from one to the other that the programmer must observe. For example, the keyword `Integer` corresponds to a C `int`, `Real` corresponds to a C `double`, and `String` corresponds to a C character pointer (`char*`). There are additional keyword-type correspondences, which we'll introduce as needed throughout this chapter.

By convention the template commands are stored in a file whose name ends with *.tm*, and the C language file produced by *mprep* ends in *.tm.c* (*.c* on DOS/Windows systems). The exact use of *mprep* is system-specific and the documentation that comes with the *MathLink* distribution for your system should be consulted for the details. Once the *.tm.c* file has been generated, it is compiled and linked with the file(s) containing the installable function, the glue code, and the *MathLink* library to produce an executable file. Once again, the details of the compilation and link step are system-dependent and can be found in the documentation that comes with your *MathLink* distribution.[8]

After the executable file has been created, the functions in it can be *installed* into the kernel using the *Mathematica* function `Install`. There are several ways to call `Install`, but the simplest way is to supply it with the name of the executable file (or its pathname, if it is located in a different directory than *Mathematica*'s current working

8. On UNIX systems, the shell script *mcc* handles these details for you.

directory[9]). The kernel will start the *MathLink* program and initiate communication with it. If it is successful, it returns a `LinkObject`.

```
link = Install["SetBit"]
LinkObject[SetBit, 5, 2]
```

You can find out what patterns are defined for a link using `LinkPatterns[`*`link`*`]`. The pattern information is transmitted to the kernel by the *mprep*-generated code during the installation process (which we'll analyze in Section 11.3.2).

These patterns were defined in the template file.

```
LinkPatterns[link]
{SetBit[n_, b_], ClearBit[n_, b_]}
```

`SetBit` and `ClearBit` can be used just as any *Mathematica* functions.

```
SetBit[1, 5]
33

ClearBit[%, 0]
32
```

Here is the *Mathematica* definition of the symbol `SetBit`:

```
?SetBit
Global`SetBit
SetBit[n_, b_] :=
  ExternalCall[LinkObject["SetBit", 5, 2],
   CallPacket[0, {n, b}]]
```

`ExternalCall` sends an expression using `LinkWrite` and waits for a reply. The above definition says that whenever the pattern `SetBit[x_, y_]` is matched, a message of the form `CallPacket[0, {n, b}]` will be sent on the link to the `SetBit` program. (The details of how normal expressions are received by a C program using the *MathLink* library will be discussed in Section 11.4.)

`ClearBit` is similar:

```
?ClearBit
Global`ClearBit
ClearBit[n_, b_] :=
  ExternalCall[LinkObject["SetBit", 5, 2],
   CallPacket[1, {n, b}]]
```

9. In the MacOS version of *MathLink*, if the executable file cannot be found in the working directory, the user is prompted to locate the file using an "open file" dialog box. On DOS/Windows systems, be sure to use double backslashes \\ to indicate a directory separator, since certain two-character sequences that begin with a backslash (e.g., \n) are interpreted as single characters inside of *Mathematica* strings.

In this case, the message sent to the server program is of the form `CallPacket[1, {n, b}]`. This is very similar to the protocol used by our manually constructed server program: The kernel sends an indication of the service being requested (in this case, `0` or `1`), followed by the arguments required for that service.

Note that no type checking is done by the kernel. Rather, the code generated by *mprep* checks each `MLGettype` call for failure to obtain an argument of the correct type from the link. It returns the symbol `$Failed` in such cases, without calling either of the installed functions.

```
SetBit[a, b]
$Failed
```

This behavior can be rather startling — most *Mathematica* functions simply do not evaluate if they are called incorrectly. To avoid this outcome, do as much type checking as possible in the link pattern. In this example, it would have been better to use a pattern like `SetBit[n_Integer, b_Integer]`. Note, however, that the call still would fail if either n or b were too large to fit into a C integer. Going even further, we could use a pattern of the form `SetBit[n_Integer /; -2^31 <= n < 2^31, ...]`, but bounds checks such as this may not be portable across processor architectures. The fact is that some types of errors can be detected only by the installed function; we'll discuss error handling in detail in Section 11.7.

When you are finished using an installed function, it's a good idea to uninstall it to recover system resources. Not surprisingly, this is done using a function called `Uninstall`. The argument to `Uninstall` is the `LinkObject` returned by `Install`. If you forgot to save this object, you can retrieve it by using the `Links[]` function, which returns a list of all active `LinkObjects`.

The first link in the list is the link from the kernel to the front end.

```
Links[]
{LinkObject[Mathematica, 1, 1],
  LinkObject[SetBit, 5, 2]}

Uninstall[%[[2]]]
SetBit
```

`Uninstall` closes the link and removes the definitions of the installed functions from the *Mathematica* session. Back in the *MathLink* program, `MLMain` returns and, after executing any code that the programmer may have put after the call to `MLMain`, the program terminates.

Incidentally, in version 3.0, if the name passed to `Install["name"]` is the name of a directory rather than a file, then the kernel will look for an executable file called *name*/`$SystemID`/*name* instead (`$SystemID` is a global symbol containing the type of system that is being used). This allows binary files for different types of machine architectures to be stored on the same computer system (e.g., a file server with heterogeneous clients), yet be referred to by a consistent name.

Exercises

1. Add the `getbit` function of Exercise 11.1.3.1 to the installable program developed in this section.
2. Use *mprep* to create an installable version of the `SubString` function of Exercise 11.1.3.2. The template file data type `String` corresponds to the C type `char*` (pointer to character).

11.2.2 Using lists as arguments

Suppose now that we want to enhance `SetBit` and `ClearBit` so that they can set or clear more than one bit at a time. Writing a C function to take a variable number of arguments is extremely tricky and sometimes system-dependent; furthermore, *mprep* can't deal with it. However, an installable function can take a *list* of `Integers` or `Reals` as an argument, which correspond to C *arrays* of `int`s or `double`s, respectively. The *MathLink* type keywords used for this purpose are `IntegerList` and `RealList`.

Here is the template for the new version of `SetBit`; the template for `ClearBit` is analogous:

`setbit` is declared to take an `IntegerList` as its second argument. Note how the pattern checks for this type.

```
:Begin:
:Function:      setbit
:Pattern:       SetBit[n_Integer, b:{__Integer}]
:Arguments:     {n, b}
:ArgumentTypes:{Integer, IntegerList}
:ReturnType:    Integer
:End:
```

The C code for the new `setbit` function is straightforward, except that certain parameter-passing conventions must be observed.

The second *and* third parameters correspond to the `IntegerList` type keyword.

```
int setbit(int n, int bits[], long nbits)
{   int i;
    for (i = 0; i < nbits; i++)
        n |= (1<<bits[i]);
    return n;
}
```

The first parameter to `setbit` corresponds to the first `Integer` argument, as before. However, *two* parameters are required to receive an `IntegerList` argument: an integer array of unspecified size, and a long integer that gives the size of the foregoing array.[10] This parameter layout is mandatory, as it corresponds to the way that the *mprep*-generated code will call the function.

10. Similarly, the template type keyword `RealList` corresponds to a pair of C arguments having types pointer-to-`double` and `long`.

The C definition of the new `clearbit` function is analogous to the new `setbit`. The glue code is the same as before and is not repeated here.

Here is an example of using this new *MathLink* program:

```
link = Install["SetBit2"]
LinkObject[SetBit2, 4, 2]
```

This sets bits 0 through 9.

```
SetBit[0, Range[0, 9]]
1023
```

This clears bits 0, 2, and 4.

```
ClearBit[63, {0, 2, 4}]
42
```

Unfortunately, using a single integer as the second argument to the new functions no longer works, because it doesn't match any of the link's patterns.

```
SetBit[32, 0]
SetBit[32, 0]

LinkPatterns[link]
{SetBit[n_Integer, b:{__Integer}],
  ClearBit[n_Integer, b:{__Integer}]}
```

It's somewhat of a disappointment that we have "broken" `SetBit` (and `ClearBit`) in the sense that calling them the old way no longer works. Clearly, we would like to be able to associate additional patterns with these functions. There's a mechanism for doing this automatically each time the link is opened, as we'll see next.

Don't forget to `Uninstall`.

```
Uninstall[link]
SetBit2
```

11.2.3 Including *Mathematica* code in a template file

We would like to fix the problem observed at the end of the last section by allowing `SetBit` to be called as either `SetBit[`*`num, bit`*`]` or `SetBit[`*`num, {bits}`*`]`. An obvious way to achieve this end would be to create a *Mathematica* definition like the following:

```
SetBit[n_Integer, b_Integer] := SetBit[n, {b}]
```

We want this definition to be created automatically when the external program is installed. This can be accomplished using the `:Evaluate:` template statement.

The `:Evaluate:` template statement allows you to specify an arbitrary *Mathematica* expression that is sent to the kernel, verbatim, during the installation process. You can use any number of `:Evaluate:` statements in a template file. In the following example we use `:Evaluate:` statements to create usage messages for `SetBit` and

`ClearBit`, and also to define a rule for each function that changes a single bit number into a list containing that number.

```
:Begin:
(... template for SetBit as before ...)
:End:
```

A single `:Evaluate:` statement can span multiple lines, as long as each line after the first begins with a space or a tab and there are no blank lines in the middle.

```
:Evaluate:      SetBit[n_Integer, b_Integer] :=
                SetBit[n, {b}]

:Evaluate:      SetBit::usage =
                "SetBit[n, b] sets the b'th bit in the
                machine-sized integer n to 1.
                SetBit[n, {b1, b2, ...}] sets bits
                b1, b2, ...";
```

```
:Begin:
(... template for ClearBit as before ...)
:End:
```

`:Evaluate:` statements can be interspersed with function templates; their order within the template file is the order in which the kernel will evaluate them.

```
:Evaluate:      ClearBit[n_Integer, b_Integer] :=
                ClearBit[n, {b}]

:Evaluate:      ClearBit::usage =
                "ClearBit[n, b] sets the b'th bit in the
                machine-sized integer n to 0.
                ClearBit[n, {b1, b2, ...}] clears bits
                b1, b2, ...";
```

If we build the program as before (without making any changes at all to the C code portion of it), this is the result:

```
link = Install["SetBit3"]
LinkObject[SetBit3, 6, 2]
```

`SetBit` now has a usage message and *two* definitions: one from the function template and another from the `:Evaluate:` statement.

```
??SetBit
SetBit[n, b] sets the b'th bit in the  machine-sized
   integer n to 1.  SetBit[n, {b1, b2, ...}] sets
   bits  b1, b2, ...
SetBit[n_Integer, b_Integer] := SetBit[n, {b}]
SetBit[n_Integer, b:{__Integer}] :=
  ExternalCall[LinkObject["SetBit3", 6, 2],
   CallPacket[0, {n, b}]]
```

As before, we can set (or clear) a set of bits.

```
SetBit[0, Range[0, 2]]
7
```

But now we also can set and clear individual bits.

```
ClearBit[%, 1]
5
```

```
SetBit[%, 5]
37
```

Always `Uninstall` when finished.

```
Uninstall[link]
SetBit2
```

The general lesson to be learned here is the following: Play to the strengths of each of your development tools. Bit operations are easier in C than in *Mathematica*, so all of that code is written in C. On the other hand, error checking is easier is *Mathematica* than in C, so it is done as much as possible using the patterns defined in the template file. Furthermore, the `:Evaluate:` statement is a powerful tool that should be exploited as much as possible: If some of the work can be done in *Mathematica* code without compromising performance, don't hesitate to include as much *Mathematica* code in the template file as you need to get the job done with the least amount of effort.

Exercise

1. Add another definition to the `SubString` *MathLink* program so that `SubString` can be called with either one or two indices: `SubString[s, n]` should have the same effect as `SubString[s, n, n]`. While you're at it, add a usage message.

11.3 Debugging *MathLink* Programs

All of the examples of link connection that we have seen in this chapter so far have used what is called a *parent-child* connection, in which the *Mathematica* kernel plays the role of the parent process and the *MathLink* program is the child process.[11] You can't debug an external program using this type of connection, since there's no way for a debugger to gain control of the child process. However, there is another type of connection, called a *peer-to-peer* connection, that allows both ends of the connection to be run independently — for example, one or both processes can be run from within a debugger. Peer-to-peer connections also can be used to interpose a third *MathLink* process between the parent and the child, which can be used to monitor and/or filter the messages that are sent between them.

11.3.1 Peer-to-peer connections

In order for a peer-to-peer connection to be made, each *MathLink* program must be told the name of a link to open. Using the TCP network protocol, a link name is an integer; using the local communication protocols on MacOS or Windows, a link name can be an arbitrary string. One side opens the link in *create*[12] mode, after which the other side

11. It is also possible for the kernel to play the role of the child process; in fact, this is exactly what happens when the kernel is started from the front end.
12. Prior to version 3.0, *create* mode was called *listen* mode.

opens it in *connect* mode. It doesn't matter which side does which, because after the link is established, there is no distinction made between the parties.

In C, the link name and mode are specified by the arguments to `MLOpenArgv` or, in an installable program, `MLMain`. You could hard-code that information into the function call, but the convention of passing the program's command-line arguments to `MLOpenArgv` or `MLMain` (as we have been doing) makes this unnecessary. For example, on a UNIX or DOS system you could run an installable *MathLink* program like this:[13]

```
setbit -linkname 2500 -linkcreate
```

and the C strings `"-linkname"`, `"2500"`, and `"-linkcreate"` will be passed to `MLMain`. `MLMain` will then pass these arguments to `MLOpen`, which will listen for a *MathLink* connection on port `2500`.

On MacOS, you merely run the *MathLink* program by double-clicking it; a dialog box will prompt you for the name of a port to create. On Windows, you can use either the command-line arguments method (by using the *Program Manager*'s **Run** command in the **File** menu) or the dialog box method.

In *Mathematica* you can connect to an existing link using the `LinkConnect` function. The `LinkObject` returned from this call is then passed to `Install`.[14]

```
Install[LinkConnect["2500"]]
```

Note that even though the port name is an integer, it must specified as a string. The link is now "open for business" and can be used in the normal way.

To debug a *MathLink* program, all that is necessary is to run the program inside of your system's debugger, using the appropriate method (command-line arguments or dialog box) to set up a peer-to-peer connection. You can then set breakpoints in the program and inspect variables as you would with any other program. Consult the documentation for your debugger for details.

Exercise

1. Use your system's debugger to step through the execution of any *MathLink* program.

---

13. Programs created with older versions of *MathLink* should be given the arguments `-linkmode listen` (two separate arguments) instead of `-linkcreate`.
14. In versions prior to 3.0 replace `LinkConnect[`*`name`*`]` by `LinkOpen[`*`name`*`, LinkMode->Connect]`. Alternatively, you can pass the `LinkMode->Connect` option directly to `Install`, which will call `LinkOpen` for you.

11.3.2 Monitoring traffic on a link

Another way to use peer-to-peer connections is to interpose a third *MathLink*-aware program between the two parties whose interaction is of interest. This third program simply opens a link to each of the other two parties and copies data back and forth between them, as shown in Figure 11-1. Each party thinks that it is connected directly to the other.

Figure 11-1 Monitoring *MathLink* traffic using a "snoop" program.

You can use such a program to generate a log of the traffic on a link in real time. This is a great way to gain an understanding of the workings of higher-level protocols (such as the one used by `Install` and *mprep*, or the one used by the front end and the kernel).

Although it would be possible to write the "snoop" program in C, it's much easier to use another *Mathematica* kernel for this purpose, because the kernel gives us such a high-level interface to a link. (Of course, we pay a price in terms of memory usage by doing this.)

To begin, start another *Mathematica* kernel[15] (which we will refer to as the *snooping kernel*) and evaluate the following definition:

Evaluate this definition in the new (snooping) kernel.

```
snoop[link1_LinkObject, link2_LinkObject] :=
Module[{msg},
    LinkConnect /@ {link1, link2};
    While[LinkConnectedQ[link1] &&
          LinkConnectedQ[link2],
        If[LinkConnectedQ[link1] && LinkReadyQ[link1],
            msg = LinkReadHeld[link1];
            If[msg === $Failed, Break[]];
            Print["--> ", HoldForm @@ msg];
            If[LinkConnectedQ[link2],
                LinkWriteHeld[link2, msg]];
        ];
        If[LinkConnectedQ[link2] && LinkReadyQ[link2],
            msg = LinkReadHeld[link2];
            If[msg === $Failed, Break[]];
            Print["<-- ", HoldForm @@ msg];
```

15. Note that you can run two kernels from the same front end, and have each one take input from a different window. If memory is really tight, try running each kernel directly, without using the front end.

```
                    If[LinkConnectedQ[link1],
                        LinkWriteHeld[link1, msg]];
                ];
            ]
        ]
```

This function may look complicated, but the overall gist of it is actually quite simple. `link1` and `link2` are links to the communicating parties, established as described in Section 11.3.1. Each time through the `While` loop, the `snoop` function checks to see if there is any input on `link1`; if so, it is read from the link, printed, and written to `link2`. The converse of this procedure is then performed on `link2`. The loop continues to execute as long as both links remain active.

The devil is in the details, as the saying goes, so here they are. `LinkConnectedQ` is a predicate that checks to see if there is another party at the other end of a link. However, right after a link is opened, `LinkConnectedQ` will return `False` until some other operation is performed on the link, such as a `LinkRead` or `LinkWrite`. The purpose of the call to `LinkConnect` at the beginning of the function is to verify the existence of the other parties without sending or receiving any data on the links, so that subsequent calls to `LinkConnectedQ` will return `True` (assuming, of course, that the links are in fact connected to the other parties).

`LinkReadyQ` polls a link and returns `True` if there are data waiting to be read from it. It is necessary to call `LinkReadyQ` before each `LinkRead` to prevent the `snoop` function from blocking on an empty link. `LinkReadHeld` is like `LinkRead` except that it wraps the expression received from the link in `Hold` before returning it. Similarly, `LinkWriteHeld` takes an expression wrapped in `Hold` and writes it to a link sans the `Hold` head.

Here is how the `snoop` function is used. First, open a connection (parent-child or peer-to-peer) to the *MathLink* program and open a peer-to-peer connection to the original kernel. Note that we use `LinkLaunch`, *not* `Install`, to connect to the *MathLink* program, because we do not wish to install any functions into *this* kernel.

Evaluate these expression in the snooping kernel.

```
server = LinkLaunch["SetBit2"];
client = LinkCreate["impostor"];
```

Next, in your original kernel, which we will refer to as the *client kernel*, connect to the link named `"impostor"` using `Install`:

Evaluate this expression in the client (original) kernel.

```
link = Install[LinkConnect["impostor"]];
```

At this point you will notice that the client kernel seems to be stuck. This behavior is expected, since `Install` is waiting for information from the server program. The situation at this moment is analogous to an open electric circuit; the circuit is "closed" using the `snoop` function (back in the snooping kernel):

Evaluate this expression in the snooping kernel.

These messages are being sent from the *MathLink* server program to the client kernel.

```
snoop[client, server]

<-- DefineExternal[SetBit[n_Integer, b:{__Integer}],
   {n, b}, 0]
<-- SetBit[n_Integer, b_Integer] :=  SetBit[n, {b}]
<-- SetBit::usage =  "SetBit[n, b] sets the b'th bit\
   in the  machine-sized integer n to 1. \
   SetBit[n, {b1, b2, ...}] sets bits  b1, b2, ...";
<-- DefineExternal[ClearBit[n_Integer,\
   b:{__Integer}], {n, b}, 1]
<-- ClearBit[n_Integer, b_Integer] :=  ClearBit[n, {b}]
<-- ClearBit::usage =  "ClearBit[n, b] sets the b'th\
   bit in the  machine-sized integer n to 0. \
   ClearBit[n, {b1, b2, ...}] clears bits  b1, b2, ...";
<-- End
```

A number of things happen: First, several messages are printed by `snoop`; second, the `Install` function in the client kernel returns; and third, now the snooping kernel is the one that seems to be stuck. The snooping kernel will remain in this state until one of the other two parties closes its link.

Now let's analyze the messages printed above. The description that follows is based on an inspection of the code generated by *mprep* and the code for `Install`. (The latter is implemented entirely external to the kernel, and hence its definition can be viewed using `??Install`.) The *mprep*-generated code does the following: For each `:Evaluate:` statement in the template file, it sends the contents of that statement, verbatim, as a string; and for each function template, it sends an expression of the form `DefineExternal[`*`pattern, arguments, index`*`]`, where *`pattern`* and *`arguments`* are taken directly from the template. On the other end of the connection, `Install` is in a loop, reading expressions from the link until the symbol `End` is encountered. If the expression is a string, it is converted to a normal expression using `ToExpression`, which then evaluates.

The *`index`* in each `DefineExternal` expression is an integer that is used internally by the *mprep*-generated code to identify the installed function that corresponds to the given pattern. The result of `DefineExternal` is a *Mathematica* function definition of the type we have already seen:

```
SetBit[n_Integer, b:{__Integer}] :=
  ExternalCall[LinkObject["impostor", 11, 3],
   CallPacket[0, {n, b}]]
```

`ExternalCall` simply writes the given `CallPacket` to the link and reads back the result. Note that the integer in the `CallPacket` that is sent from the kernel to the server is the same as the one that was sent from the server to the kernel in the `DefineExternal` expression.

Now here are some examples of what is sent across the link when an installed function is called:

Evaluate this in the client kernel's window.

```
SetBit[0, {1, 3, 5}]
```

These messages appear in the snooping kernel's window.

```
--> CallPacket[0, {0, {1, 3, 5}}]
<-- 42
```

This result appears in the client kernel's window.

```
42
```

Client kernel input.

```
ClearBit[%, 3]
```

Snooping kernel output.

```
--> CallPacket[1, {42, {3}}]
<-- 34
```

Client kernel result.

```
34
```

When you are done experimenting, clean things up by uninstalling the link in the client kernel:

Client kernel input.

```
Uninstall[link]
impostor
```

Back in the snooping kernel, the snoop function returns and the following message may appear:

```
LinkObject::linkd:
   LinkObject[impostor, 11, 3]
     is closed; the connection is dead.
```

You still need to close the snooping kernel's link to the server program manually.

Snooping kernel input.

```
LinkClose[server]
```

The server program will then terminate.

Exercises

1. Use the snoop function to monitor the traffic between the kernel and your substring extraction program.
2. The technique illustrated in this section can also be used, with some slight modifications, to monitor the interaction between the *Mathematica* front end and the kernel. When a *Mathematica* kernel is being run from a front end, the kernel is said to be in *MathLink mode*, and the global variable `$ParentLink` contains the link that connects the kernel to the front end:

```
$ParentLink
LinkObject[Mathematica, 1, 1]
```

As long as `$ParentLink` contains the name of a connected link, the kernel will "take orders" from the program at the other end of that link. If `$ParentLink` is

`Null`, however, the kernel interacts directly with the user through its own input/output window. Try the following experiment:

```
$ParentLink = Null
```

After you type this, your front-end window will "hang." If you pay close attention, you will notice that a new window has appeared on the screen: the kernel's input/output window. You can reestablish *MathLink* mode by typing the following in that window:

```
$ParentLink = First[Links[]]
```

The kernel's window will now "hang" and the following output will appear in the original front-end window (note that the output number will not match the input number):

```
LinkObject[Mathematica, 1, 1]
```

By assigning a different link to `$ParentLink`, you can make the kernel think that any other program is its front end. In particular, if `$ParentLink` is a link to a kernel running the `snoop` program, you can monitor the traffic between the kernel and the front end. Here are the details.

First, start a kernel from the front end and evaluate the definition of the `snoop` function in that kernel. We will refer to this kernel as the "snooping" kernel. Next, evaluate the following[16] in the kernel you wish to monitor (which we will henceforth refer to as the "child" kernel):

```
$ParentLink = LinkCreate["child"]
```

The child kernel's window will hang. Evaluate the following in the snooping kernel:

Adopt-a-kernel. The snooping kernel is now the child kernel's "parent."

```
kernel = LinkConnect["child"]
LinkObject[child, 2, 2]
```

At this point the snooping kernel should have two open links. The first is the link to its own front end.

```
Links[]
{LinkObject[Mathematica, 1, 1],
 LinkObject[child, 2, 2]}
```

Short-circuit the two links using `snoop`.

```
snoop @@ Links[]
```

From this point on, anything typed into the snooping kernel's front-end window is sent to the child kernel without being interpreted by the snooping kernel (as you'll note by examining the `In/Out` numbers). In between each input and its correspond-

16. In versions prior to 3.0, use `LinkOpen[`*`name`*`, LinkMode->Listen]` instead of `LinkCreate[`*`name`*`]`.

ing output, however, snoop prints all of the packets that are transmitted. This is an excellent way to learn about the protocol used by the front end and the kernel.

Try entering calculations that produce printed output, error messages, or graphics. Try using `Interrupt[]` and `Input[]`. If you are using a version of the front end that has a function browser, try using it to look up some definitions. Experiment!

When you are done with this exercise and you want to salvage your original session, evaluate the following to disconnect the child kernel from the snooping kernel:

This is typed into the snooping kernel's front-end window, but is evaluated by the child kernel.

```
$ParentLink = Null;
```

Alternatively, if your child kernel was originally started by the front end, you can reconnect it to its front-end window by evaluating

```
$ParentLink = First[Links[]];
```

After evaluating either of these expressions, the snooping kernel's front-end window will hang, because that kernel is still stuck inside the snoop routine. To get snoop to terminate, you must switch to the child kernel's window and close the link back to the snooping kernel.

Child kernel input.

```
LinkClose[kernel];
```

You will see the usual error message and then an input prompt from the snooping kernel.

11.4 Manual Data Handling

In many common situations, you can rely on *mprep* to handle all of the details of communicating with the kernel. Up to this point we have seen examples of (or at least heard mention of) the types `Integer`, `Real`, `String`, `Symbol`, `IntegerList`, and `RealList`. In addition, *mprep* supports — this is an undocumented feature! — the types `ShortInteger`, `LongInteger`, `Float`, `Double`, and `LongDouble`, with the obvious correspondence to C types.[17] Version 3.0 introduces several more types (see Section 11.9.3) that are used for 16-bit character data. You can use any of these types for function arguments, and any of the scalar numeric types, `String`, or `Symbol` for function return values.

17. [Wolfram 96] §A.11 states that `MLPutReal` and `MLGetReal` are "normally equivalent" to `MLPutDouble` and `MLGetDouble`, whatever that means. Furthermore, the *MathLink Reference Guide* [WRI 93c] coyly points out that this equivalence is not guaranteed in future versions of *MathLink!*

However, there are many situations that can't be handled by *mprep*. For example, *mprep* doesn't allow an installable function to return an `IntegerList` or `RealList`, because these types are passed as two values — a pointer to data and a length — and C functions can return only a single value. Hence, such functions have to bypass the *mprep*-generated code and return their results directly to the kernel. Section 11.4.1 shows how to do this. Then in Section 11.4.2 we extend the technique to returning arbitrary *Mathematica* expressions. Finally, in Section 11.4.3 we address the inverse problem, i.e., accepting arbitrary expressions as arguments to an installable function.

11.4.1 Returning lists

Suppose that we want to create functions that pack and unpack a list of binary values into an arbitrary-length bit vector. (Such functions would have been very handy to have when we developed the Huffman coding example of Section 5.3.5.) The inputs *and* outputs to these functions will be lists of integers. However, the only return types for which *mprep* can generate code are scalars and character strings. In order to return an array of integers (or doubles), the programmer must bypass the *mprep*-generated code and make calls to low-level *MathLink* functions directly.

For every type supported by *MathLink* there are C library functions called `MLPut`*`type`* and `MLGet`*`type`*. For example, to send an `Integer` to the kernel, you could call `MLPutInteger`. In most cases this is handled by the *mprep*-generated code — for example, if your template file declares the return type of a function as `Integer`, then the *mprep* code will store the return value from the function in a C `int` and pass this integer to `MLPutInteger` for you. However, when you need to return a type that *mprep* doesn't support, you have to make these calls yourself. If you make an explicit `MLPut`*`type`* call to return a value, your function should not return any value to the *mprep* code. Therefore, the C function is declared `void`.

In the present example we wish to return an `IntegerList` to the kernel, which is done in C with the `MLPutIntegerList` library function.

Here is the C code that implements `packbits`. The format of a bit vector is a list of integers, where the first integer in the list indicates the total number of valid bits. Bits are packed a maximum of 8 per integer. While this is not the most compact representation possible, the kernel could easily convert these "byte codes" to a character string[18] using `FromCharacterCode`; the result could be written to a file using `WriteString` or the functions in `Utilities`BinaryFiles``.

```
int bitsPerWord = 8;
```

18. The reason for not using character strings in the first place is that a 0 byte in the middle of a C string would be interpreted as the end of the string. See Section 11.9.3, "String and symbol types," for two ways around this problem.

The function returns no value. The two arguments constitute an `IntegerList`.

Allocate storage for the return value.

First word is the number of valid bits.

Pack the bits.

Send the return value to the kernel.

Free the storage.

```
void packbits(int bits[], long nbits)
{   long nints = (nbits + bitsPerWord - 1) /
                 bitsPerWord + 1;
    int *bitvec = (int*) calloc(nints, sizeof(int));
    int i;

    bitvec[0] = nbits;

    for (i = 0; i < nbits; i++) {
        bitvec[i / bitsPerWord + 1] |= (bits[i] <<
            (bitsPerWord - 1 - i % bitsPerWord));
    }

    MLPutIntegerList(stdlink, bitvec, nints);

    free(bitvec);
}
```

The salient features of this code are as follows. First, `packbits` is declared `void`, i.e., it does not return a value to the *mprep* code. Second, `MLPutIntegerList` is used to send the result directly to the kernel, bypassing the *mprep* code. Third, note that it is the programmer's responsibility to free any dynamically allocated storage.

The first argument to `MLPutIntegerList` is a link descriptor. The *mprep* code stores the descriptor for the link to the kernel in a global variable called `stdlink`. The remaining two arguments to `MLPutIntegerList` are a pointer to the data and the size of the `IntegerList` — the type correspondence is the same for return values as it is for function arguments.

It is vitally important to free any dynamically allocated storage after you are finished using it; `MLPut`*type* will not do it for you! Remember that the *MathLink* program is initialized only once, when the link is opened, and continues to run for as long as the link is open. Therefore, if you forget to release dynamically allocated storage, then the program will have a "memory leak" — i.e., it will lose some amount of memory *every time* the installed function is used. Eventually the *MathLink* program will run out of memory and chaos will ensue.

In the template file, you must declare the return type as `Manual` so that the *mprep* code will not expect your C function to return a value to it. Here are the template file statements for the function `packbits`.

This predicate is used for type-checking the pattern below.

```
:Evaluate:      BinaryQ[x_] := x == 0 || x == 1

:Evaluate:      PackBits::usage = "PackBits[list] packs
 a list of binary digits into a list of byte codes."

:Begin:
:Function:      packbits
```

```
:Pattern:       PackBits[bitlist_] /;
                      VectorQ[bitlist, BinaryQ]
:Arguments:     {bitlist}
:ArgumentTypes:{IntegerList}
:ReturnType:    Manual
:End:
```

The return type is `Manual`.

Next, we need to write a function called `unpackbits` that reverses the action of `packbits`. The C code for `unpackbits` follows the same general paradigm as that for `packbits`: Allocate storage for the return value, do the "bit bashing," call `MLPutIntegerList`, and free the temporary storage.

```
void unpackbits(int bitvec[], long nints)
{   int nbits = bitvec[0];
    int *bits = (int*)malloc(nbits*sizeof(int));
    int i;

    for(i = 0; i < nbits; i++)
        bits[i] = (bitvec[i / bitsPerWord + 1] >>
            (bitsPerWord - 1 - i % bitsPerWord)) & 1;

    MLPutIntegerList(stdlink, bits, nbits);

    free(bits);
}
```

Here are the template file statements for `unpackbits`:

More type-checking predicates.

```
:Evaluate:      ByteCodeQ[x_] :=
                    IntegerQ[x] && 0 <= x < 256
:Evaluate:      BitVectorQ[{f_Integer,
                          r__?ByteCodeQ}] := True /;
                 8*(Length[{r}]-1) < f <= 8*Length[{r}]
:Evaluate:      BitVectorQ[___] := False

:Evaluate:      UnpackBits::usage =
 "UnpackBits[bitvec] extracts the
 individual binary digits from a bit vector."
```

The rest of the template is nearly identical to that for `packbits`.

```
:Begin:
:Function:      unpackbits
:Pattern:       UnpackBits[bitvec_?BitVectorQ]
:Arguments:     {bitvec}
:ArgumentTypes:{IntegerList}
:ReturnType:    Manual
:End:
```

Note how thoroughly the argument to `unpackbits` is checked for validity. This is done to avoid having to do error checking in the C code. It isn't the error checking per se

that's so much harder; the difficulty lies in trying to return a meaningful result when an error is discovered. (We'll deal with this problem in Section 11.7.)

Here is an example of using this *MathLink* program:

```
link = Install["bitvec"]
LinkObject[bitvec, 6, 2]
```

The usage message works as expected.

```
?PackBits
PackBits[list] packs a list of binary  digits into a
   list of byte codes.
```

Here are some test data.

```
bits = Table[Random[Integer], {18}]
{0, 1, 0, 1, 1, 1, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1}
```

`PackBits` returns a list of integers. The first integer is the number of valid bits.

```
bv = PackBits[bits]
{18, 93, 32, 64}
```

`UnpackBits` returns the original data.

```
UnpackBits[bv]
{0, 1, 0, 1, 1, 1, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1}
```

Exercise

1. Write a *MathLink* function that complements a specified range of bits in a bit vector. If no range argument is supplied, the function should complement the entire bit vector.

11.4.2 Returning arbitrary normal expressions

In the previous section we learned how to send a list of integers manually to the kernel. Because this is such a common thing to want to do, there is a special *MathLink* library call, `MLPutIntegerList`, for doing it. (Likewise, there is a corresponding library call for putting a `RealList`.) In general, however, in order to send a normal expression to the kernel, you have to put it one part at a time.

As mentioned earlier, there are `MLPut`*type* functions for atomic types such as `Integer`, `Real`, `String`, and `Symbol`. But in order to send a normal expression, you need to use the `MLPutFunction` call to inform the kernel that what is being sent is the head of a normal expression. `MLPutFunction` takes two arguments in addition to a link: a character string, which is interpreted as the symbolic head of the expression, and a long integer, which tells the kernel how many parts the expression will have.[19] If you specify a part count of *n*, for example, then you must follow the call to `MLPutFunction` with *n* calls to other `MLPut`*type* functions.

19. To send expressions whose heads are not symbols (e.g., `Derivative[2][f][x]`) you need to use the `MLPutNext` and `MLPutArgCount` functions. See [Wolfram 96] §2.12.12 or the *MathLink Reference Guide* [WRI 93c] for details.

For example, suppose that you need to send a list of mixed types to the kernel. There is no function provided for this, so you have to construct the list from lower-level calls. You could do it as follows:

This sends the list {"one", 2, 3., Pi} to the kernel.

```
MLPutFunction(stdlink, "List", 4);
MLPutString(stdlink, "one");
MLPutInteger(stdlink, 2);
MLPutReal(stdlink, 3.0);
MLPutSymbol(stdlink, "Pi");
```

Of course, expressions can be nested, and so can calls to MLPutFunction. Here is how you could send the expression N[expr /. x->2.5] to the kernel:

The indentation of the code is intended to convey the nesting of the subexpressions within the overall expression.

```
MLPutFunction(stdlink, "N", 1);
  MLPutFunction(stdlink, "ReplaceAll", 2);
    MLPutSymbol(stdlink, "expr");
    MLPutFunction(stdlink, "Rule", 2);
      MLPutSymbol(stdlink, "x");
      MLPutDouble(stdlink, 2.5);
```

This code puts onto the link the expression N[ReplaceAll[expr, Rule[x, 2.5]]], which is the internal form of N[expr /. x->2.5].

An alternative to putting every part of a large expression separately is to format the entire expression as a string, and put the expression ToExpression[*string*] instead. Used in combination with C's sprintf library function, this is a very easy way to construct and send complicated expressions to the kernel:

```
char s[100];
char *expr = "BesselJ[0, x]/(x^2 + 1)";
char *var = "x";
double value = 2.5;
sprintf(s, "N[%s /. %s->%f]", expr, var, value);
MLPutFunction(stdlink, "ToExpression", 1);
MLPutString(stdlink, s);
```

As an application of manual return values, we will modify the packbits function so that instead of returning a flat list it returns an "abstract data type" of the form BitVector[*nbits*, {*bytecode1*, *bytecode2*, ...}]. The purpose of this is to give bit vectors their own head so that they may be more easily recognized by other *Mathematica* code.

The template file statements (except for the usage message) are identical to those for the previous version of packbits, because both versions take the same arguments and return their results manually. The C code is also identical except that the call to MLPutIntegerList is replaced by the following sequence of calls:

head + 2 parts
first part
second part

```
MLPutFunction(stdlink, "BitVector", 2);
  MLPutInteger(stdlink, nbits);
  MLPutIntegerList(stdlink, data, nwords);
```

Note that the second part of the `BitVector` is another normal expression, namely, a list of integers. Rather than send the list part by part, however, we simply use `MLPutIntegerList`. Here is a demonstration of the new version of `packbits`:

```
link = Install["bitvec2", LinkMode->Connect]
LinkObject[bitvec2, 7, 2]

bv = PackBits[bits]
BitVector[18, {93, 32, 64}]
```

Obviously, `unpackbits` is going to have to be modified to take a `BitVector` as an argument. However, this is much easier than you might suspect: Simply destructure the argument in the `:Pattern:` statement and specify the extracted parts as the `:Arguments:`. The advantage of doing things this way is that the C code for this version of `unpackbits` is *identical* to that of the previous version! Here are the relevant template file statements:

```
:Evaluate:      BitVectorQ[BitVector[f_Integer,
                          {r__?ByteCodeQ}]] := True /;
                 8*(Length[{r}]-1) < f <= 8*Length[{r}]
:Evaluate:      BitVectorQ[___] := False

:Begin:
:Function:      unpackbits
:Pattern:       UnpackBits[b:BitVector[nbits_, data_]]
                /; BitVectorQ[b]
:Arguments:     {Prepend[data, nbits]}
:ArgumentTypes:{Integer, IntegerList}
:ReturnType:    Manual
:End:
```

This is an excellent example of the reason for the `:Arguments:` field: There are three pattern variables in the pattern (`b`, `nbits`, and `data`), but only two of these are passed to the *MathLink* function.

The new `unpackbits` works on `BitVector` objects.

```
UnpackBits[BitVector[18, {93, 32, 64}]]
{0, 1, 0, 1, 1, 1, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1}
```

When finished, uninstall.

```
Uninstall[link]
bitvec2
```

Exercises

1. Create an installable function called `StringTokenize` that breaks up a string into *tokens* (substrings that are separated by one or more characters from a set of *token delimiters*). For example,

The second argument is a string containing the token delimiters.

```
StringTokenize["A one, and a two.", " ,."]
{"A", "one", "and", "a", "two"}
```

You can use the C library function `strtok`. Add `StringTokenize` to your substring extraction program.

2. Define additional patterns for `StringTokenize` in the template file so that the second argument can be specified as a list of individual characters.

11.4.3 Manual arguments

Just as an installable function can manually send *Mathematica* expressions to the kernel, so can it also manually receive them as arguments. This may be necessary if the type of argument to the function is not supported by *mprep*. For each `MLPut`*type* function there is a corresponding `MLGet`*type* function, the only significant difference being that the data parameters to an `MLGet`*type* are passed by reference (e.g., `MLGetInteger` takes a *pointer* to a C integer as its second argument).

Suppose that we want to write a version of `unpackbits` that could accept a `BitVector` as an argument directly (i.e., without being destructured first). Here are the relevant template file entries:

The argument is not destructured.

Note that the argument type is now `Manual`.

```
:Begin:
:Function:      unpackbits
:Pattern:       UnpackBits[bitvec_?BitVectorQ]
:Arguments:     {bitvec}
:ArgumentTypes: {Manual}
:ReturnType:    Manual
:End:
```

You might well wonder, if the argument type is `Manual`, why does the `:Arguments:` statement still specify an argument? The `:Arguments:` statement (along with the `:Pattern:` statement) is used to construct the *Mathematica* definition for `UnpackBits`. In other words, it tells the kernel what data to send across the link. The kernel doesn't know or care what is done with those data on the other end of the link. The `:ArgumentTypes:` statement, on the other hand, is used by *mprep* to construct the code that receives the data from the kernel. If the keyword in this statement is `Manual`, *mprep* simply *doesn't write* that code. Note, however, that if you want to send manifestly typed arguments as well as manual arguments to the same *MathLink* function, the `Manual` keyword must be the last element of the `:ArgumentTypes:` statement. Otherwise, *mprep* would have no way of telling which arguments to generate code for and which ones to ignore.

Here is the C code for the new `unpackbits`. The main difference from the previous version is, of course, the explicit calls to various `MLGet`*type* functions. There also is an important extra bit of cleanup: a call to `MLDisownIntegerList` (explained below).

There are *no* arguments to unpackbits.

These are just like MLPut-*type* calls except the data parameters are passed by reference.

Important extra cleanup step!

```
void unpackbits( /* Manual */ )
{   long count, nbits, nwords;
    int *data, *bits;

    MLCheckFunction(stdlink, "BitVector", &count);
    MLGetLongInteger(stdlink, &nbits);
    MLGetIntegerList(stdlink, &data, &nwords);

    bits = (int*)malloc(nbits * sizeof(int));
    /* ...bit bashing - same as before... */

    MLDisownIntegerList(stdlink, data, nwords);

    MLPutIntegerList(stdlink, bits, nbits);
    free(bits);
}
```

MLCheckFunction(*link*, *func*, &*nargs*) checks the link to see that the head of the expression on the link is *func*, and if so, it sets *nargs* to be the number of arguments to that function. MLCheckFunction is useful in functions like unpackbits, in which the expression on the link always has a certain head. (If we did not know what head to expect, we could have used MLGetFunction(*link*, &*func*, &*nargs*) to get both the name of the function and the number of arguments to it. However, MLGetFunction requires an extra cleanup step, as explained next.)

When you read a fixed-size data type such as an integer from a link, you pass a pointer to an integer to MLGetInteger. However, when you read a variable-sized data type such as an integer list, you pass a pointer to an *integer pointer* to MLGetIntegerList. The crucial difference is that in the first case, all of the storage needed by the integer (4 bytes) already exists. However, in the second case, the only storage that has been allocated is for the pointer to the first integer in the array, but *not the array itself*. The *MathLink* library allocates this storage and sets the integer pointer that you pass to MLGetIntegerList to point at it. You should not try to deallocate this storage yourself (as was done with the bits array); in fact, the *MathLink* documentation discourages programs from even writing into this storage. Instead, you must inform *MathLink* that it's okay to deallocate the storage by using MLDisownIntegerList. Note that when you declare an argument of type IntegerList in a template file, *mprep* generates the code that calls MLDisownIntegerList. But since the argument in this case is declared as Manual, you need to make this call before your installable function returns.

Returning to MLCheckFunction and MLGetFunction for a moment: After calling MLGetFunction(*link*, &*func*, &*nargs*), you are required to free the memory being used by *func* by calling MLDisownSymbol(*link*, *func*). The advantage of MLCheckFunction is that it obviates the need to disown any storage.

Forgetting to call MLDisown*type* when required is another possible source of memory leaks!

Remember to exploit the strengths of each development tool at your disposal! If you find yourself putting and getting a lot of expressions manually, then you need to ask yourself why. Chances are you're doing symbolic processing in C that you should be doing in *Mathematica* template code.

11.5 Integrating Installable Functions and Packages

It is easy to imagine circumstances in which one would like to include installable functions in a package. Todd Gayley [Gayley 94c] points out that there are two completely different ways of doing this, each with its own strengths and weaknesses. One way is to use `:Evaluate:` statements to embed the entire package inside of the template file; there are examples of this in the *MathLink* documentation. The other way, which Gayley advocates, is to call `Install` from within an ordinary *Mathematica* package file. In the following two sections we will discuss each of these methods.

11.5.1 Embedding a package in a template file

Suppose that we want to extend the `setbit` and `clearbit` functions of Section 11.2.1 to work on bit vectors. One way to do this would be to modify the code for these functions to handle the `BitVector` type as an argument and as a return value. However, there really is no need to do so: Given the number of the bit to be modified, it is quite easy to figure out which element of the bit vector data structure contains the relevant bit. By passing only this single integer we not only simplify the C code (the original definitions of `setbit` and `clearbit` from Section 11.2.1 can be used), we also cut down on the amount of data passed between the kernel and the installed function — dramatically so in the case of very large bit vectors.

We can define the following "wrapper" function for `setbit` (the wrapper function for `clearbit` is analogous):

`SetBit` extracts the relevant byte code and calls `setbit`. The return value is passed to `ReplacePart` to update the `BitVector`'s list of bits.

```
SetBit[b:BitVector[nbits_, data_], n_Integer] /;
    BitVectorQ[b] && 1 <= n <= nbits :=
  With[{p = Quotient[n - 1, 8] + 2},
    BitVector[nbits,
      ReplacePart[data,
        setbit[data[[p]], Mod[n - 1, 8] + 1],
        p]
    ]
  ]
```

Now we have a situation in which one function implements a public interface to another function — a situation that cries out for a package structure. We can use `:Evaluate:` statements to embed within the template file all of the necessary calls (Section 8.2.1) to `BeginPackage`, `EndPackage`, and so forth, as well as the wrapper functions themselves! Here is the overall structure of the template file:

Create the package context.

```
:Evaluate:    BeginPackage["BitVector`"];
```

Usage messages for exported symbols.

```
:Evaluate:    BitVector::usage = ...
:Evaluate:    PackBits::usage = ...
:Evaluate:    UnpackBits::usage = ...
:Evaluate:    SetBit::usage = ...
:Evaluate:    ClearBit::usage = ...
```

Private subcontext starts here.

```
:Evaluate:    Begin["`Private`"];
```

(These predicates could be made public or private.)

```
:Evaluate:    BinaryQ[x_] := x == 0 || x == 1
:Evaluate:    ByteCodeQ[x_] := ...
:Evaluate:    BitVectorQ[BitVector[...]] := ...
:Evaluate:    BitVectorQ[___] := False
```

Template statements for installable functions are inside the private context.

```
:Begin:
:Function:    packbits
...
:End:

...
:Function:    unpackbits
...
:Function:    setbit
...
:Function:    clearbit
...
```

The exported function `SetBit` calls the private function `setbit`.

```
:Evaluate:    SetBit[b:BitVector[nbits_, data_],
                n_Integer] /;
              BitVectorQ[b] && 1 <= n <= nbits :=
              With[{p = Quotient[n - 1, 8] + 1},
                BitVector[nbits,
                  ReplacePart[data,
                    setbit[data[[p]], Mod[n-1,8]+1],
                  p]
                ]
              ]
```

`ClearBit` is analogous to `SetBit`.

```
:Evaluate:    ClearBit[....] := ....
```

End of private subcontext.

```
:Evaluate:    End[];
```

Protect exported symbols.

```
:Evaluate:    Protect[{BitVector, PackBits,
                UnpackBits, SetBit, ClearBit}];
```

End of package.

```
:Evaluate:    EndPackage[];
```

Starting with a fresh kernel, here is how the installation of this *MathLink* program affects the state of the user's session:

Tabula rasa...

```
?Global`*
Information::nomatch:
   No symbol matching Global`* found.
```

Install the *MathLink* program.

```
link = Install["bitvec4"]
LinkObject[bitvec4, 4, 2]
```

The only global symbol is the link itself.

```
?Global`*
Global`link
```

```
link = LinkObject["bitvec4", 4, 2]
```

These five symbols have been exported from the package.

```
?BitVector`*
BitVector   ClearBit    PackBits    SetBit     UnpackBits
```

The rest of the package's symbols — in particular, the installed functions `setbit` and `clearbit` — are private.

```
?BitVector`Private`*
BitVector`Private`b
BitVector`Private`BinaryQ
BitVector`Private`bitlist
BitVector`Private`bitvec
BitVector`Private`BitVectorQ
BitVector`Private`ByteCodeQ
BitVector`Private`clearbit
BitVector`Private`data
BitVector`Private`n
BitVector`Private`nbits
BitVector`Private`newbit
BitVector`Private`p
BitVector`Private`p$
BitVector`Private`setbit
BitVector`Private`x
```

Unfortunately, there are some problems uninstalling, since we have protected two symbols defined in link patterns.

```
Uninstall[link]
Unset::write: Tag PackBits in TooBig is Protected.
Unset::write:
   Tag UnpackBits in UnpackBits[<<1>>] is Protected.
bitvec4
```

The error messages triggered by `Uninstall` would have been prevented if we had created wrapper functions for *all* of the installed functions and protected only the wrapper functions.

The advantage to integrating installable functions and packages in this way is that there are fewer files to keep track of. In fact, you can even embed the C code for the installable functions in the template file; *mprep* simply passes C code through to the *.tm.c* file. Of course, this also complicates the development process, as any change to the package code, the C code, or the templates forces a recompilation of the entire *MathLink* program! It may be advantageous to keep the various components separate during the development process, and combine them only for purposes of distribution.

But the main disadvantage of this technique is that it requires that users learn a new paradigm for loading a package — namely, calling `Install` instead of `Needs`.

11.5.2 Calling Install from within a package

The second way to integrate installable functions and packages is to call `Install` from within an ordinary *Mathematica* package. Although this may seem like the most obvious thing to do, there are a number of subtle difficulties involved.

Here is an illustration of what happens when `Install` is used within a package (*note:* the following example was created using a "fresh" kernel session):

Start a package.

```
BeginPackage["test`"]
test`
```

Install a *MathLink* program.

```
link = Install["bitand"]
LinkObject[bitand, 2, 2]
```

The symbol `BitAnd` referenced in this statement is not the one created by `Install`.

```
Context[BitAnd]
BitAnd::shdw:
   Warning: Symbol BitAnd
     appears in multiple contexts {test`, Global`};
     definitions in context test`
     may shadow or be shadowed by other definitions.
test`
```

The `BitAnd` function was installed in the `Global`` context.

```
?Global`BitAnd
Global`BitAnd

Global`BitAnd[Global`x_, Global`y_] :=
  ExternalCall[LinkObject["bitand", 2, 2],
   CallPacket[0, {Global`x, Global`y}]]
```

Exit the package context.

```
EndPackage[];
```

The symbol `link` was created in the package context.

```
Uninstall[test`link]
bitand
```

The problem demonstrated above is a consequence of the fact that every installed function is created in the `Global`` context, no matter where `Install` is called from. This happens because `Install` explicitly overrides the value of `$Context`. Thus, in order to embed an installable function in a package such as the following,

```
BeginPackage["pkgname`"]
... (other package stuff) ...
Install["externalprog"];
... (other package stuff) ...
EndPackage[]
```

the corresponding template file must enter and leave the package's context using `Begin...End`:

```
:Evaluate:      Begin["pkgname`"];
... (other template statements)...
:Evaluate:      End[];
```

Forearmed with this knowledge, it is possible to embed calls to `Install` within a package, which the user can then load in the familiar way. However, the main disadvantage to this approach is that it requires the distribution and installation of (at least) two files rather than one. The user has to place the *MathLink* program in a directory where it will be found by `Install`, or else he may have to edit the call to `Install` within the package file to reflect the *MathLink* program's location on his particular system.

This problem can be mitigated in version 3.0 by using the following technique: If the argument to `Install` is of the form `"name`"`, the kernel will search all of the directories in `$Path` for a file named *name*. This allows one to place both of the files (the package file and the *MathLink* program file) in any directory in `$Path` and everything will work transparently.

11.6 Callbacks to the Kernel

Sometimes during the course of execution of an installed function the function needs to have the *Mathematica* kernel evaluate an expression. This is accomplished by calling the function `MLEvaluate` (which is generated by *mprep*). `MLEvaluate` takes a string as an argument, which it sends to the kernel for evaluation. The value returned by the kernel is retrieved using `MLGettype` calls, just as would be done for manual arguments (Section 11.4.3).

As an example we will create a *MathLink* function that finds a root of a continuous function of a single variable (henceforth referred to as the *subject function*) on a specified interval using bisection. The bisection logic will be implemented in C, but each time the subject function needs to be evaluated, a request will be made to the kernel to do so. In this way, the subject function can depend on other symbols and functions in the *Mathematica* session that the installed function doesn't know about.

The pattern for the installed function will be `Bisect[expr_, {x_Symbol, x0_, x1_}]` (by analogy with the built-in function `FindRoot`). It is expected that the endpoints of the interval `x0` and `x1` are numerical, that `expr` evaluated at each endpoint is numerical, and that `expr` has opposite signs at the two endpoints. All of this is checked by *Mathematica* code attached to the pattern as a condition. Before this code calls the installed function, it converts `expr` to a string in order to obviate the need for the installed function to accept an arbitrary normal expression as an argument.

```
:Begin:
:Function:      bisect
```

```
:Pattern:      Bisect[expr_, {x_Symbol, x0_, x1_}] /;
 NumericQ[x0] && NumericQ[x1] &&
 With[{y0 = N[expr /. x->x0], y1 = N[expr /. x->x1]},
  NumberQ[y0] && NumberQ[y1] && Sign[y0]Sign[y1] == -1
 ]
:Arguments:    {ToString[InputForm[expr]],
                x, N[x0], N[x1]}
:ArgumentTypes:{String, Symbol, Real, Real}
:ReturnType:   Real
:End:
```

Before passing the arguments, `expr` is converted to a string and `x0` and `x1` are evaluated numerically.

Now for the C code. Here is the code for the main `bisect` routine, which depends on another routine, `evaluate_expr`, to evaluate the *Mathematica* expression.

```
double bisect(char *expr, char *x,
              double x0, double x1)
{   double xmid, y0, y1, ymid;
    char *buf = malloc(strlen(expr) + strlen(x) + 100);

    y0 = evaluate_expr(buf, expr, x, x0);
    y1 = evaluate_expr(buf, expr, x, x1);

    while (fabs(x1 - x0) > .000001)
    {   xmid = (x0 + x1)/2.0;
        ymid = evaluate_expr(buf, expr, x, xmid);

        if (sign(ymid) == sign(y0))
            x0 = xmid, y0 = ymid;
        else
            x1 = xmid, y1 = ymid;
    }

    free(buf);

    return (fabs(y0) < fabs(y1)) ? x0 : x1;
}
```

`buf` is used by `evaluate_expr`.

`expr` is evaluated at each endpoint.

The midpoint is calculated, and the expression is evaluated there.

The bisection step.

Free `buf`.

Return whichever endpoint is closer to the root.

Here is the code for `evaluate_expr`. It uses the C library function `sprintf` to construct the string `"N[expr /. x->xval]"` and sends it to the kernel for evaluation using `MLEvaluate`. It then retrieves the result using `MLGetReal`.

```
double evaluate_expr(char *buf, char *expr,
                     char *x, double xval)
{   double result;
    sprintf(buf, "N[%s/.%s->%f]", expr, x, xval);
    MLEvaluate(stdlink, buf);
    MLNextPacket(stdlink);
    MLGetReal(stdlink, &result);
    return result;
}
```

The `MLNextPacket` call requires some explanation. Data are sent on a *MathLink* link in the form of *packets*, which are simply *Mathematica* expressions having special heads. The packet heads alert the receiver to the type of data that is arriving. For example, the kernel sends a `CallPacket` to a *MathLink* program to request that an installed function be called. `MLEvaluate` sends an `EvaluatePacket` to the kernel, which informs the kernel that the packet is *not* the return value from the installed function, but rather a request for an evaluation. The `MLNextPacket` function simply advances to the next packet on the link and returns the type of that packet. We are taking it on faith that the next packet will be a reply to our evaluation request (a `ReturnPacket`), and that the expression contained within it is a real number — neither of which are necessarily the case if an error occurs during the evaluation. We'll address that possibility in the next section.

Here is a demonstration of `bisect`:

```
link = Install["bisect"];

??Bisect
Bisect[expr, {x, x0, x1}] searches for  a root of expr
   in the interval x0<x<x1. expr must have opposite
   signs  at the endpoints of the interval.

Bisect[expr_, {x_Symbol, x0_, x1_}] /;
   NumericQ[x0] && NumericQ[x1] &&
    With[{y0 = N[expr /. x -> x0],
      y1 = N[expr /. x -> x1]},
     NumberQ[y0] && NumberQ[y1] &&
      Sign[y0]*Sign[y1] == -1] :=
  ExternalCall[LinkObject["bisect", 9, 2],
   CallPacket[0, {ToString[InputForm[expr]], x,
     N[x0], N[x1]}]]
```

This call calculates an approximation to `Sqrt[2]`.

```
Bisect[x^2 - 2, {x, 1, 2}]
1.41421
```

The pattern check prevents calls like this from being made (the expression has the same sign at both endpoints).

```
Bisect[x^2 - 2, {x, 2, 3}]
Bisect[-2 + x^2, {x, 2, 3}]
```

The expression can depend on other symbols since the kernel, not the *MathLink* program, is evaluating it.

```
Bisect[Sin[Pi x], {x, 1/2, 5/4}]
1.
```

Note that as an alternative to constructing a C string and passing it to `MLEvaluate`, you can send an expression to be evaluated using a sequence of `MLPut`*type* calls (Section 11.4.2). If you decide to go this route, you *must* wrap the entire expression in the special head `EvaluatePacket` so the kernel doesn't think the expression it is

receiving is the return value from the installed function! Here is the requisite sequence of MLPut*type* calls for the current example:

```
MLPutFunction(stdlink, "EvaluatePacket", 1);
  MLPutFunction(stdlink, "N", 1);
    MLPutFunction(stdlink, "ReplaceAll", 2);
      MLPutFunction(stdlink, "ToExpression", 1);
        MLPutString(stdlink, expr);
      MLPutFunction(stdlink, "Rule", 2);
        MLPutSymbol(stdlink, x);
        MLPutDouble(stdlink, xval);
```

This code puts the expression `EvaluatePacket[N[ReplaceAll[ToExpression[expr], Rule[x, xval]]]]` onto the link. Note the trick of wrapping `expr`, which is a string, inside `ToExpression`. About the only reason to go to all this trouble is if the expression you are sending to the kernel is being built as it is being sent. Otherwise, using `sprintf` and `MLEvaluate` seems like the easier course of action.

Exercise

1. Use the `snoop` function of Section 11.3.2 to observe the exchange of packets during a call to `Bisect`.

11.7 Error Checking

There is a limit to how "bulletproof" we can make *MathLink* functions through type checking in the link patterns. There are some cases in which error conditions will arise during the execution of the function no matter how carefully (within reason) the arguments are checked. Here's an example of such an error occurring during the execution of `bisect`:

```
Bisect[1/x, {x, -1, 1}]

                                    1
Power::infy: Infinite expression ---- encountered.
                                    0.

$Failed

Uninstall[link];
```

The error message shown above was generated by the kernel during the evaluation of `1/x /. x->0`. The kernel returned a packet containing the expression `DirectedInfinity[]` in response. What happened next is that `evaluate_expression` tried to get a real number out of the packet, which caused an error. The value of the variable `result` was thus unchanged by the call to `MLGetDouble(stdlink, &result)`, and the value returned by `evaluate_expr` was the value of an uninitialized local variable! The bisection process continued, albeit to an incorrect conclusion (`bisect` attempted to return 1).

The reason the symbol `$Failed` was returned is a bit more complicated. It turns out that after an error occurs on a link, *all subsequent calls using that link continue to fail until the error condition is cleared explicitly.* The error condition wasn't noticed until `bisect` returned to the *mprep*-generated code. That code (a) cleared the error condition and (b) returned the symbol `$Failed`.

In other words, we were just *lucky* that something reasonable happened. Since `bisect` sanguinely ignores link errors, every call to `evaluate_expr` after the first bad one returned garbage, too — it seems almost miraculous that `bisect` terminated at all! In general we won't be so lucky; the *MathLink* function might return a nonsensical result, loop forever, or turn into a pillar of digital salt (i.e., crash).

11.7.1 General error checking

The least we should do in a situation like the one above is to print a reasonable error message and return the symbol `$Failed` immediately. In order to return a value other than a real number, however, we will have to put the return value manually. We already know how to do that: Simply change the `:ResultType: Real` statement to `:ResultType: Manual` in the template file, and use `MLPut`*type* functions to return values from `bisect`. Here is the new `bisect`:

No return value.

```
void bisect(char *expr, char *x, double x0, double x1)
{   double xmid, y0, y1, ymid;
    char *buf = malloc(strlen(expr) + strlen(x) + 100);

    if (evaluate_expr(buf, expr, x, x0, &y0) ||
      evaluate_expr(buf, expr, x, x1, &y1)) {
        MLPutSymbol(stdlink, "$Failed");
        return;
    }

    while (fabs(x1 - x0) > .000001)
    {   xmid = (x0 + x1)/2.0;
        if (evaluate_expr(buf, expr, x, xmid, &ymid))
        {   MLPutSymbol(stdlink, "$Failed");
            return;
        }
        /*** bisect step - as before ***/
    }
    free(buf);

    MLPutReal(stdlink,
              (fabs(y0) < fabs(y1)) ? x0 : x1);
}
```

`evaluate_expr` now returns a status code.

If it fails, return `$Failed`.

If `evaluate_expr` fails, bail out.

Normal termination: Use `MLPutReal` to return the answer.

Now for the error checking itself. As the saying goes, it's a dirty job, but .... This code is rather subtle and we'll go through it step by step below.

Now the return value is a status code and the result of the evaluation is a by-reference parameter.

Request evaluation as before.

Throw away packets until a RETURNPKT is received.

Attempt to read a Real.

MathLink error?
Construct an error message.

Clear the link status.
Discard the current packet.
Send the error message and ignore the reply.

```
int evaluate_expr(char *buf, char *expr, char *x,
                  double xval, double *result)
{   int p, err;

    sprintf(buf, "N[%s/.%s->%f]", expr, x, xval);
    MLEvaluate(stdlink, buf);
    while ((p = MLNextPacket(stdlink)) &&
           (p != RETURNPKT))
        MLNewPacket(stdlink);
    MLGetReal(stdlink, result);

    if (err = MLError(stdlink)) {
        sprintf(buf,
                "Message[Bisect::linkerror \"%.70s\"]",
                MLErrorMessage(stdlink));
        MLClearError(stdlink);
        MLNewPacket(stdlink);
        MLEvaluateString(stdlink, buf);
    }

    return err;
}
```

The `while` loop after the call to `MLEvaluate` keeps throwing away packets until either a `RETURNPKT` is received or an error occurs (in which case `MLNextPacket` returns 0). This is a robust way to wait for a result from the kernel. The `MLNewPacket` function throws away the remainder of the current packet without getting any more data out of it. In the code above we call `MLNewPacket` each time we want to ignore a packet.

If an error occurs while waiting for the `RETURNPKT`, the `MLGetReal` call following the `while` loop faces certain failure. However, there is no need to check for errors until after the call to `MLGetReal`, as explained next.

The `if` statement following the call to `MLGetReal` checks for the occurrence of a *low-level MathLink* error. This type of error usually indicates that the parties on either side of the link have somehow gotten "out of step" with each other — e.g., attempting to get a `Real` out of a packet containing a normal expression. However, there are many other sources of low-level errors and, in general, *any MathLink* library call can return an error status.[20] Checking the status of *every* library call results in hard-to-read nested `if` statements, liberal use of `goto`, or other strange contrivances (inspect any *.tm.c* file for examples). None of this is really necessary, however, since low-level *MathLink*

20. To be precise, most *MathLink* library calls return 0 to indicate that an error occurred. The exact nature of the error can be determined by calling `MLError`.

errors disable the link until they are cleared with `MLClearError`. Thus, error checks need to be made only at certain strategic points in the program, because once an error occurs, additional library calls can do no further damage.

One such strategic point is just before `evaluate_expr` returns. If any *MathLink* call up to this point failed, `MLError` will indicate the reason for it. `evaluate_expr` then assumes that the result of the kernel evaluation is garbage and constructs an error message. The message `Bisect::linkerror` is defined in the template file as `"Low-level MathLink error: ``1``"`. The library function `MLErrorMessage` returns a string describing the most recent error on the link. The link status is cleared using `MLClearError`, and finally an attempt is made to deliver the error message to the kernel. Here, `MLEvaluateString` is used, which is like `MLEvaluate` except that it ignores the kernel's reply.[21] No attempt is made to see if the evaluation of the `Message` succeeds, since there would be little that could be done if it failed.

Note that `MLClearError` is called *after* the error message is constructed but *before* the error message is sent. This is the only correct way to do it: After `MLClearError` has been called, you can no longer find out what the error was; conversely, before `MLClearError` has been called, you can't send the error message because the link will ignore you.

Here is an example of the behavior of the new and improved `bisect`.

```
Bisect[1/x, {x, -1, 1}]

                                          1
Power::infy: Infinite expression  -- encountered.
                                          0.

Bisect::linkerror:
   Low-level MathLink error: MLGet? out of sequence

$Failed
```

This message is generated by the kernel.

This message is generated by `evaluate_expr`.

This return value is sent by `bisect`.

While it may seem that all that has been accomplished is the generation of a cryptic error message, the real point of this modification is that `bisect` returns `$Failed` immediately after detecting an error. Recall that in the previous version, `bisect` continued to labor toward an incorrect answer, and it was only after `bisect` returned that the *mprep*-generated code noticed the link error and returned `$Failed`.

11.7.2 Checking the types of packets and expressions

The code given in the previous section will handle any conceivable error situation, but it leaves something to be desired with regard to user-friendliness! For the `bisect` func-

21. `MLEvaluateString` is new in version 3.0. Users of earlier versions should use `MLEvaluate`, followed by a loop that "eats" everything up to and including the next `RETURNPKT`.

tion it would be nice to check specifically for the type of error we have been encountering since it is likely to occur occasionally.

We can use MLGetType to find out what type of expression is in a packet before attempting any MLGet*type* call(s) on that packet. If the RETURNPKT does not contain a real number, we will issue a more specific error message. Here, then, is the entire solution for evaluate_expr:

```
int evaluate_expr(char *buf, char *expr, char *x,
                  double xval, double *result)
{   int p, err;

    sprintf(buf, "N[%s/.%s->%f]", expr, x, xval);
    MLEvaluate(stdlink, buf);

    while ((p = MLNextPacket(stdlink)) &&
           (p != RETURNPKT))
        MLNewPacket(stdlink);

    if (p == RETURNPKT) {
        if (MLGetType(stdlink) == MLTKREAL)
            MLGetReal(stdlink, result);
        else {
            MLNewPacket(stdlink);

            sprintf(buf,
                  "Message[Bisect::plnr, %s, %s, %f]",
                   expr, x, xval);
            MLEvaluateString(stdlink, buf);
            if (!MLError(stdlink)) return MLEUSER;
        }
    }

    if (err = MLError(stdlink)) {
        sprintf(buf,
                "Message[Bisect::mlink, \"%.70s\"]",
                MLErrorMessage(stdlink));
        MLClearError(stdlink);
        MLNewPacket(stdlink);
        MLEvaluateString(stdlink, buf);
    }

    return err;
}
```

Request evaluation as before.

Wait for a RETURNPKT.

If the packet contains a Real, get it.

Otherwise, it's an error.

Throw away the packet.

Issue the infamous plnr error message (see below).

Ignore the reply.

Return an error status.

Check for low-level *MathLink* errors. This code is the same as before.

When the returned expression is not a real number, the code constructs and evaluates a call to *Mathematica*'s built-in Message function that issues a plnr error. Users of *Mathematica*'s plotting functions probably will recognize this error:

```
Message[Bisect::plnr, "1/x", "x", 0.]
Bisect::plnr:
   1
   - is not a machine-size real number at x = 0..
   x
```

The code then returns the predefined constant MLEUSER, which is an error number that is guaranteed to be higher than any of *MathLink*'s own error numbers. You can define as many of your own error codes as you like using MLEUSER, MLEUSER + 1, etc.

Note that if any other kind of *MathLink* error occurs, it will be caught by the same error-checking code that we used in our previous version of evaluate_expr. So, for example, if something dreadful goes wrong during the attempt to issue the Bisect::plnr message, a more general (and probably cryptic) *MathLink* error message will be issued.

Finally, the fruit of our labors:

```
Bisect[1/x, {x, -1, 1}]
```

This message is generated by the kernel.

```
                                        1
Power::infy: Infinite expression ------ encountered.
                                        0.
```

This message is generated by evaluate_expr.

```
Bisect::plnr:
   1
   - is not a machine-size real number at x = 0..
   x
```

This return value is sent by bisect.

```
$Failed
```

11.8 Making Installed Functions Abortable

Any installable function that performs a nontrivial amount of computation should be made *abortable*, particularly if the function involves any kind of iteration. What makes this possible is a global variable called MLAbort in the *MathLink* program that is set to 1 *asynchronously*[22] by the *mprep* code if the kernel sends an interrupt packet (which it does in response to the user typing the keyboard combination that requests an abort). Therefore, any time-consuming loop should be structured as follows:

```
while(test && !MLAbort) {
    /* loop body */
}
```

22. This is strictly true only on operating systems that have preemptive multitasking, such as UNIX, OS/2, and Windows 95. MacOS System 7.*x* and Windows 3.*x* programmers need to take some extra precautions, discussed in [Gayley 94c].

Note that your function continues to execute even after MLAbort has been set; it is *your* responsibility to do something about it. If you don't, then when your function eventually returns (*if* it ever does, that is), the *mprep* code will ignore its return value and return the symbol $Aborted instead. [Gayley 94c] points out that this probably should not be allowed to happen: If the user makes a call like *f*[*installed-func*[*args*]] then the result of aborting the computation will be *f*[$Aborted], rather than $Aborted (which is what a computation that does not involve any installed functions would return). He recommends instead that installable functions return the normal expression Abort[] when an abort request occurs, which causes a computation to abort all the way to the top level. Compare the following two alternatives:

```
f[$Aborted]
f[$Aborted]

f[Abort[]]
$Aborted
```

This behavior can be achieved by giving installable functions the Manual return type and structuring their C code as shown below:

```
func(args)
{
    while (test && !MLAbort) {
        /* loop body */
    }

    if (MLAbort)
        MLPutFunction(stdlink, "Abort", 0);
    else
        MLPuttype(stdlink, the_answer);
}
```

A head with no parts (Abort[]).

11.9 Miscellaneous *MathLink* Data Types

Mathematica can handle some types of data for which there are no counterparts in C. Two immediate examples are exact integers and arbitrary-precision numbers, which can have far more digits than can be represented by any of the numeric types in C. *MathLink* solves this problem by allowing programs to put and get numbers in textual, rather than binary, form. This technique will be discussed in Section 11.9.1.

Version 2.2 added the ability to put and get multidimensional arrays of numbers with a single library call. The array types will be discussed in Section 11.9.2.

Finally, version 3.0 adds a few new types for dealing with 16-bit character data, which will be discussed in Section 11.9.3.

11.9.1 Numbers as text

As mentioned at the beginning of Section 11.4, there are *MathLink* functions for putting and getting `short` or `long` `int`s, `float`s, `double`s, and `long double`s. These functions have names like `MLPutShortInteger`, `MLGetFloat`, etc.

Conspicuously absent from this list are functions for putting and getting *unsigned* integer types. This really is a problem only for the type `unsigned long`, since any smaller unsigned integers can be put or gotten using a larger signed integer type. A general mechanism for getting around this problem is to put or get numbers in textual, rather than binary, form. This technique can be used to get and put numbers that have many more digits than can be represented by native C types (although it's not at all clear what you would *do* with such numbers inside of a C program).

For example, `MLGetLongInteger` will fail if the integer on the link is greater than $2^{31}-1$. But `MLGetString` will succeed, returning the integer as a string (e.g., `"1234567890"`). If the integer is within the range of an `unsigned long` ($0..2^{32}-1$), then it can be extracted from the string using the standard C library function `sscanf`. The following C code fragment illustrates these steps:

```
char *s;
unsigned long x;
if(MLGetNext(stdlink) == MLTKINT) {
    MLGetString(stdlink, &s);
    sscanf(s, "%lu", &x);
    MLDisownString(stdlink, s);
}
else {
    /* error - do something about it */
}
```

`MLGetNext` returns a constant indicating the type of data on the link (`MLTKINT` signifies an integer, `MLTKREAL` signifies a real number, and so on). However, the fact that there is an integer on the link does not mean that it can fit into any C integer type. Production-quality code would, of course, check for this. (Don't assume that `sscanf` does so!)

Conversely, an integer can be put as a string as shown below:[23]


```
char s2[20];
sprintf(s2, "%lu", x);
MLPutNext(stdlink, MLTKINT);
MLPutString(stdlink, s2);
```

23. This particular technique does not work in earlier versions of *MathLink*. The *textual interface* functions, `MLPutData` and `MLGetData`, can be used to send and receive any type of data in textual form. However, those functions are now obsolete, and their use is strongly discouraged.

In this case, you must use `MLPutNext` to tell the *MathLink* library what type of data is contained in the string.

11.9.2 Array types

Beginning in version 2.2, *MathLink* provides functions for putting and getting multidimensional arrays of any of the scalar numeric types. In addition to an argument that specifies a pointer to the data, these functions take an array of dimensions, rather than a scalar length; an array of strings giving the heads to use in each dimension; and the number of dimensions. For example, the following C code fragment sends a 2×3 matrix of short integers to the kernel:

```
short int data[2, 3] = {{1, 2, 3}, {4, 5, 6}};
long ndims = 2;
long dims[2] = {2. 3};
MLPutShortIntegerArray(
    stdlink, data, dims, NULL, ndims);
```

The fourth parameter to `MLPut`*type*`Array` is a list of character strings that specify the names of the heads to use in each dimension of the array. Passing a `NULL` pointer causes the head `List` to be used for all dimensions of the array.

The following C code fragment shows how to use the `MLGet`*type*`Array` functions, using `MLGetLongDoubleArray` as a specific example:

```
long double *data;
long *dims;
char **heads;
long ndims;
MLGetLongDoubleArray(
    stdlink, &data, &dims, &heads, &ndims);
/* ...use the data... */
MLDisownLongDoubleArray(
    stdlink, data, dims, heads, ndims);
```

Note that all of the arguments (except the link) are passed by reference, and that the `MLGet`*type*`Array` function allocates the necessary storage for each of the C arrays (`data`, `dims`, and `heads`). When you are finished using the array, you should be sure to call the matching `MLDisown`*type*`Array` function to allow the *MathLink* library to reclaim that storage.

Finally, note that in order to use the array data you must calculate your own indices, because the dimensions of the array were unknown when the code was compiled. For example, the (2, 3) element of a 10×10 array `x` would be accessed as `x[2 * 10 + 3]`.

11.9.3 String and symbol types

Mathematica strings may contain 16-bit characters. The *MathLink* `String` type encodes all characters outside of a certain range — *including some 8-bit characters* — using multibyte sequences. In version 2.2, *MathLink* programs must manipulate such data with explicit translation functions (see the note at the end of this section). Version 3.0 introduces several new data types and functions for dealing with 16-bit character data in a more direct fashion.

The following string contains some special characters.

```
s = "Q\\s\n~üπ"
Q\s
~üπ
```

`\\` is a *Mathematica* escape sequence for the backslash character. The `\n` escape sequence represents a newline character. ü is a standard character in every Macintosh font; methods for entering such characters may vary from system to system. π can be entered in a system-independent way in version 3.0 using the escape sequence `\[Pi]`.

Here is the kernel's internal representation of `s`.

```
ToCharacterCode[s]
{81, 92, 115, 13, 126, 252, 960}
```

In order to gain some insight into the *MathLink* representation of strings, we will use the following installable function that takes a string, as passed to it by *mprep* code, and returns a list of all of the 8-bit integers it finds in that string. (You should write the *mprep* template file for this function as an exercise; note that it puts its result manually.)

```
void stringpeek(char *s)
{   long len = strlen(s);
    int *charcodes = (int*) calloc(len, sizeof(int));
    int i;

    for (i = 0; i < len; i++)
        charcodes[i] = s[i];

    MLPutIntegerList(stdlink, charcodes, len);
    free(charcodes);
}
```

Below, `stringpeek` is installed into the kernel under the name `StringPeek`, and is then used on the string `s` defined earlier.

Install `stringpeek`.

```
link = Install["string"];
```

Here are the character codes passed by *MathLink*. Users of version 2.2 will see a different result.

```
StringPeek[s]
{81, 92, 92, 115, 92, 48, 49, 53, 126, 92, 51, 55, 52,
 92, 58, 48, 51, 67, 48}
```

And here are the individual characters corresponding to those codes.

```
FromCharacterCode /@ % // InputForm
{"Q", "\\", "\\", "s", "\\", "0", "1", "5", "~",
 "\\", "3", "7", "4", "\\", ":", "0", "3", "C", "0"}
```

Comparing this result to the original string shows that printable ASCII characters such as `Q`, `s`, and `~` are represented as themselves. There is only one exception to this rule: A backslash is represented as a double backslash (each `\\` in the result of `FromCharacterCode` is a single backslash formatted in `InputForm`). The newline character is represented by the four-character sequence `\015`; the number following the backslash is the octal (base-8) representation of the code for a newline ($15_8 = 13_{10}$). Similarly, `\374` represents character code 252_{10}. Finally, the 16-bit π character is represented using the *hexadecimal* escape sequence `\:03C0`. These escape sequences, which also can be used to input special characters in a platform-independent way, are documented in [Wolfram 96] §2.7.7. Once again, note that versions of *MathLink* earlier than 3.0 used a different method of string encoding.

mprep uses `MLGetString` to obtain `String` arguments to installable functions, so the observations that follow apply to any *MathLink* program that uses `MLGetString` or `MLPutString`, whether template-based or not.

If the only strings that a *MathLink* program sends to the kernel are constructed from printable ASCII characters (except for backslash) and/or strings received from the kernel (the `bisect` function of Section 11.7.1 is an example of such a program), then the program does not need to worry about string encodings.

On the other hand, problems can arise if a *MathLink* program compares strings received from the kernel to C strings, or if the program sends C strings to the kernel. Even a simple call like `MLPutString(stdlink, "\n")` will not work as intended. *Caveat programmer!*

For the common case in which all of the characters in the string are 8-bit quantities, you can request that *MathLink* send or receive an unencoded string using the functions `MLPutByteString( MLINK` *`link`*`, unsigned char *`*`s`*`, long` *`len`* `)` and `MLGetByteString( MLINK` *`link`*`, unsigned char **`*`s`*`, long *`*`len`*`, long` *`spec`* `)`. The *`len`* argument gives the length of the byte string explicitly, since a 0 byte may be data, rather than an indication of the end of the string. The *`spec`* argument (`MLGetByteString` only) is a character code that will be substituted for any 16-bit characters that are received.

The corresponding *mprep* type keyword is `ByteString`. The C parameters corresponding to a `ByteString` argument are `unsigned char*` and `long`.[24] `ByteString` cannot be used as a return type because the length of the string has to be passed in addi-

24. The `spec` parameter is an input to `MLGetByteString` rather than an output, so it is not passed to the installable function. The value of `spec` used by *mprep* is `'\0'`; this can be changed by editing the *.tm.c* file.

tion to the data. An installable function that returns a `ByteString` must specify `:ReturnType: Manual` in the template file and call `MLPutByteString` directly.

The following variation on the `stringpeek` function is declared in the template file to take a `ByteString` argument; other than that, and the fact that the length of the string is received as a parameter, its implementation is identical to that of `stringpeek`.

```
void bstringpeek(unsigned char *s, long len)
{   int *charcodes = (int*) calloc(len, sizeof(int));
    /* rest of function is identical to stringpeek */
}
```

```
ByteStringPeek[s]
{81, 92, 115, 13, 126, 252, 0}
```

Each 8-bit character in the string is represented by its *Mathematica* character code; the 16-bit character is replaced with the value of the *spec* argument to `MLGetByteString`.

For functions that work directly on 16-bit character strings, there is yet another string type, `UnicodeString`, and corresponding *MathLink* calls for getting and putting it. Like `ByteString`, the length of a `UnicodeString` is passed separately from the data, so `UnicodeString` can be used as an *mprep* argument type but not as a return type. When it is used as an argument, the corresponding C parameters are `unsigned short*` and `long`. Here is the `stringpeek` function once again, this time with a `UnicodeString` argument.

All characters are passed to the *MathLink* program without any encoding.

```
UnicodeStringPeek[s]
{81, 92, 115, 13, 126, 252, 960}
```

Don't forget to uninstall the string functions.

```
Uninstall[link];
```

Bear in mind that when a `ByteString` or `UnicodeString` is used as a `Manual` argument (Section 11.4.3), the storage used by the string must be released using `MLDisownByteString` or `MLDisownUnicodeString`, respectively.

Note that symbol names in version 3.0 also may contain 16-bit characters. There are six more new functions, `MLGetByteSymbol`, `MLPutByteSymbol`, `MLDisownByteSymbol`, `MLGetUnicodeSymbol`, `MLPutUnicodeSymbol`, and `MLDisownUnicodeSymbol`, that are analogous to their `String` counterparts.

The preceding discussion applies only to version 3.0. Users of version 2.2 must pass all strings as the `String` type. Note that the encoding of strings in version 2.2 is different than it is in version 3.0. For compatibility, version 2.2 programs should use the `MLPutCharToString` function to encode 16-bit characters for transmission to *Mathematica*, and use a combination of the `MLforString` macro and the `MLStringChar` function to decode strings received from *Mathematica*. These functions did not appear

until version 2.2, so they will not be found in the *MathLink Reference Manual* [WRI 93c]. Examples of their use are included in the *string.c* file on the supplementary diskette (which also contains all of the functions developed in this section).

Exercise

1. Modify any of the `bitvector` programs from Section 11.4 so that the packed bits are sent across the link in the form of a `UnicodeString`, 16 bits per character.

11.10 Additional Resources

In this chapter we have concentrated mainly on using *MathLink* to install externally compiled functions into the *Mathematica* kernel. *MathLink* also can be used by an external program to control a *Mathematica* kernel in a master-slave relationship — the *Mathematica* front end uses *MathLink* in this way. This topic will be fully explored in a future companion volume to this book. The protocol used by the *Mathematica* front end to control the kernel is investigated thoroughly in [Wagner 96b].

The official documentation for *MathLink* version 3.0 appears in [Wolfram 96] §2.12. Documentation for earlier versions is scattered all over the place. The official, albeit somewhat dated, documentation for version 2 is the *MathLink Reference Guide* [WRI 93c]. Another source, usually overlooked, is the WRI technical report entitled *Major New Features in Mathematica Version 2.2* [WRI 93a], which marks the first appearance of non-ASCII string manipulation (Section 11.9.3), array functions (Section 11.9.2), and other features such as new packet types, yield functions, and loopback links (not discussed here). The documentation for version 3.0 subsumes all of this material, of course.

Todd Gayley of WRI has written a tutorial guide to *MathLink* [Gayley 94c] that is in some ways superior to the official sources (but you will still need [WRI 93c] as a reference). Although it predates version 3.0, this tutorial is highly recommended for readers seeking an alternative presentation. Gayley's article describing his *MathLink* program for accessing binary files [Gayley 94b] also contains quite a bit of tutorial material about *MathLink*.

Some things about *MathLink* can be learned only by examining the many sample programs that come with the *MathLink* distribution. Particularly useful examples in this regard are *factorinteger2.c* and *factorinteger3.c*, which show how to read arbitrary expressions from a link.

Finally, *always* read the release notes that come with the *MathLink* distribution that you are using! In addition to containing platform-specific and compiler-specific information, they are the source of many useful tidbits about new experimental functions, alternatives to obsolete functions, and so forth.

Part 5
Miscellanea

12

Input/Output

12.1 File and Directory Management

Before we begin doing input and output to files, we ought to make sure that we are using those files that we intend to! *Mathematica* provides several functions for querying and changing the program's working directory.

`Directory[]` returns the working directory that a *Mathematica* session is using.[1]

```
Directory[] // InputForm
"Sartorius:Mathematica 3.0"
```

The current working directory is the one containing the *Mathematica* program files. We probably don't want to be writing a bunch of files into this directory, as we might overwrite something important, so we should change the working directory.

`SetDirectory` changes the working directory.[2]

```
SetDirectory["Vastus:temp"]
Vastus:temp
```

Mathematica keeps track of the directories that have been visited.

```
DirectoryStack[]
{Sartorius:Mathematica 3.0}
```

1. The pathname shown is a MacOS-style pathname, which uses : as the directory separator. On DOS/Windows systems the directory separator character is \, and on UNIX it is /. Pathnames cannot contain spaces on either of those systems.
2. On the MacOS version of the front end, one can use the **Action/Prepare Input/Paste File Pathname** menu command to locate the desired directory using the standard open file dialog box. This doesn't actually change directories; it merely pastes a pathname at the notebook's current insertion point — e.g., inside a `SetDirectory` command.

It would be a good idea to change your working directory to some scratch directory before proceeding. You can always return to the most recent directory using the `ResetDirectory[]` command.

12.2 High-Level Output

12.2.1 Put and Get

`Put[`*`expr`*`, "`*`filename`*`"]`, which can be abbreviated *`expr`* `>>` *`filename`*, writes an expression to a file.

This puts the result of the `Expand` into the file *tmp*.

```
Expand[(1 + x)^6] >> tmp
```

You can view the contents of a file using the `!!` operator. Note, however, that this is not a *Mathematica* function — it prints the contents of the file as a side effect and does not return a value.

```
!!tmp
1 + 6*x + 15*x^2 + 20*x^3 + 15*x^4 + 6*x^5 + x^6
```

`Put` overwrites whatever is already in the file, if anything. If you want to append an expression to a file, used `PutAppend` (`>>>`) instead.

The output of this command is appended to *tmp*.

```
Expand[(1 + x)^4] >>> tmp
!!tmp
1 + 6*x + 15*x^2 + 20*x^3 + 15*x^4 + 6*x^5 + x^6
1 + 4*x + 6*x^2 + 4*x^3 + x^4
```

You can read a file of *Mathematica* expressions that you created with `Put` by using `Get["`*`filename`*`"]`, which can be abbreviated `<<`*`filename`*. `Get` attempts to evaluate every line in the file as *Mathematica* input; it returns the value of the last expression read from the file.

```
<<tmp
                2    3    4
1 + 4 x + 6 x  + 4 x  + x
```

You can use `Get` to read in any file containing *Mathematica* input, including packages. However, it is better to use `Needs` to read a package, since it prevents the same package from being loaded more than once (Section 8.2.2).

12.2.2 Exporting data using special output formats

The advent of the notebook interface has made using `Put` and `Get` to save and restore *Mathematica* expressions pretty much unnecessary, if not obsolete. In this section we explore how to save data in other formats.

You may on occasion need to export data generated by *Mathematica* to some other application, for example, to do some number crunching or to create special types of charts that *Mathematica* doesn't support. It is worthwhile to think about how to export data in formats such as comma- or tab-delimited text. It's much more difficult than you might expect!

Suppose we have a table of data that we wish to export, one line per row of the table.

Here are the data to be exported.

```
data = Table[{x, Exp[x]}, {x, 1., 5.}]
{{1., 2.71828}, {2., 7.38906}, {3., 20.0855},
  {4., 54.5982}, {5., 148.413}}
```

Unfortunately, `Put` is designed to write *Mathematica* expressions in a form that allows them to be reread as input to *Mathematica*:

```
TableForm[data] >> tmp
!!tmp
TableForm[{{1., 2.718281828459045},
   {2., 7.389056098930651}, {3., 20.08553692318767},
   {4., 54.59815003314424}, {5., 148.4131591025766}}]
```

The solution to this problem is to explicitly convert the table to `OutputForm` and *then* `Put` it.

It looks pretty good, except for those blank lines.

```
OutputForm[TableForm[data]] >> tmp
!!tmp
1.   2.71828

2.   7.38906

3.   20.0855

4.   54.5982

5.   148.413
```

We can eliminate the blank lines with an option to the `TableForm` command.

```
OutputForm[TableForm[data, TableSpacing->{0}]] >> tmp
!!tmp
1. 2.71828
2. 7.38906
3. 20.0855
4. 54.5982
5. 148.413
```

Problem solved, right? Wrong! If there are numbers in scientific notation in the data, the mantissas and exponents will wind up on separate lines — and there is no easy way to associate them with each other. This makes the exported data useless:

```
edata = Table[{x, Exp[x]}, {x, 12., 17.}];
OutputForm[TableForm[edata, TableSpacing->{0}]] >> tmp
!!tmp
12. 162755.
13. 442413.
                6
14. 1.2026 10
                 6
15. 3.26902 10
                  6
16. 8.88611 10
                  7
17. 2.4155 10
```

The solution to this problem is to convert the numbers to `FortranForm` (or `CForm`) before outputting them. In this form, numbers in scientific notation print as ###.###e## (where each # is a base-10 digit).

```
Map[FortranForm, edata, {2}]
{{12., 162754.7914190039}, {13., 442413.3920089206},
  {14., 1.202604284164777e6},
  {15., 3.269017372472111e6},
  {16., 8.88611052050787e6},
  {17., 2.41549527535753e7}}
```

Here is the complete solution to the problem.

```
OutputForm[
    TableForm[
        Map[FortranForm, edata, {2}],
        TableSpacing->{0}
    ]
] >> edatafile
!!edatafile
12. 162754.7914190039
13. 442413.3920089206
14. 1.202604284164777e6
15. 3.269017372472111e6
16. 8.88611052050787e6
17. 2.41549527535753e7
```

It would probably be handy to encapsulate this logic into a new function:

Note that the pattern used allows the rows of the table to have different numbers of elements.

```
Clear[SpaceDelimitedForm]
SpaceDelimitedForm[x:{{__}..}] :=
    OutputForm[TableForm[Map[FortranForm, x, {2}],
        TableSpacing->{0}]]
```

We will demonstrate a few more special output formats in the next section. A full discussion of them is presented in [Wolfram 91] §2.7 or [Wolfram 96] §2.8.

Exercises

1. What would happen if the level specification were omitted from the `Map` function call in `SpaceDelimitedForm`?
2. Suppose you had a program that required comma-delimited text as input. Write a rule that takes a table of the form accepted by `SpaceDelimitedForm` that inserts the string `","` between every pair of expressions in the table. Then create a new function called `CommaDelimitedForm` that produces comma-delimited output.

   Are you satisfied with the way the output looks? All of those extra spaces might confuse some programs. Use `TableSpacing` to eliminate the extra spaces.
3. Now suppose you have *another* program that requires *tab*-delimited text as input. This is getting tedious, isn't it? Create a new function called `DelimitedForm` that takes an option of the form `Delimiter->`*string* . The default value for the `Delimiter` option should be a single space. Naturally, a user should be able to change the default delimiter using `SetOptions` (Section 9.2.1).

   Note: A tab can be embedded in a string using the meta-character `\t`. See [Wolfram 91] §2.8.1–2.8.2 or [Wolfram 96] §2.7.5–2.7.8 for a discussion of how to enter other special characters, including those from foreign-language character sets.

12.2.3 Save and DumpSave

Sometimes you want to save the definition of a symbol (or group of symbols) to a file, in a form that can be used to "reconstitute" the symbol at a later time. The `Save` function writes the definition of one or more symbols, along with the definitions of all the symbols they depend upon, into a file.

```
a = 5;
b := a;
```

Save doesn't save just the values of symbols, it saves their literal definitions.

```
Save["tmp", b]
!!tmp
b := a
a = 5
```

`Save` is most useful when you want to save the definitions of a large number of symbols at once. You can specify a list of symbols, a string pattern (e.g., `"*foo*"`), or a context name as the second argument to `Save`.

Starting with version 3.0, you also can save symbol definitions in a binary format using `DumpSave`. By convention, such files have the suffix *.mx*. (Regardless of the suffix, `Get` automatically figures out if a file being read is in the binary format.) The advantage of `DumpSave` over `Save` is that the binary data are much quicker to load into the kernel than the `InputForm` that is written by `Save`.

Furthermore, `DumpSave` can save the definition of an entire package using the syntax `DumpSave[`*file*`, "`*context*`"]`. Loading a binary package created by `DumpSave`

has an effect that is identical to reading in a *.m* package file — for example, the context name is added to `$Packages` and is prepended to `$ContextPath`. (See Section 8.2 for a full discussion of how packages affect the state of a *Mathematica* session.) `Save[file, "context`"]` does not have the same effect: It saves the definitions of every symbol in the context, but reading those definitions back in recreates only the individual symbols, not the context itself.

The disadvantage of `DumpSave` is that the binary format is different on each of the various computer systems that *Mathematica* supports. However, if `Get[name]` discovers that *name* is the name of a directory rather than a file, then `Get` looks for a file called *name*`/$SystemID/`*name.mx* (`$SystemID` is a global symbol containing the type of system that is being used). This allows binary packages for different types of machine architectures to be stored on the same computer system (e.g., a file server with heterogeneous clients), yet be referred to by a consistent name.

12.3 Low-Level Output

In some cases you may need more control over the format of the output than `Put` affords. The functions described in this section give you that control.

12.3.1 Streams

In *Mathematica*, a *stream* is a source of input or output. Streams can be files, *pipes* (on some operating systems), or even character strings.[3] Before a stream can be used, it must be *opened*, and when it is no longer needed, it should be *closed*.

Here's a file with some data in it.

```
"Hello there" >> tmp
!!tmp
"Hello there"
```

`OpenWrite` associates an output stream with a file.

```
s = OpenWrite["tmp"]
OutputStream[tmp, 8]
```

`OpenWrite` wipes out whatever data were in the file.

```
!!tmp
```

`OpenAppend` is similar to `OpenWrite`, with the obvious difference.

The value returned by `OpenWrite` or `OpenAppend` is passed as an argument to the functions that actually write the data to, and close, the stream. You can get a list of all open streams using the `Streams` command.

3. This is similar to *string streams* in C++ ([Stroustrup 91] §10.5.2). Associating a stream with a character string in *Mathematica* is done with the `StringToStream` function ([Wolfram 91] §2.10.8 or [Wolfram 96] §2.11.9).

The first two streams in the result were opened automatically at start-up.

```
Streams[] // InputForm
{OutputStream["stdout", 1], OutputStream["stderr", 2],
  OutputStream["tmp", 8]}
```

When you are finished with a stream, be sure to close it.

```
Close[s];
```

12.3.2 Writing to a stream

Once you have opened a stream for writing or appending, you can write data to it. There are two functions for this: `Write`, which writes *Mathematica* expressions, and `WriteString`, which writes character strings.

`Write[s, expr1, ...]` writes the given expression(s) to the stream `s`, followed by a newline character. `Write` leaves no spaces or other delimiters between the expressions.

```
s = OpenWrite["tmp"];
Clear[a, b]
Write[s, (a + b)^2, " expands to ", Expand[(a + b)^2]]
!!tmp
(a + b)^2" expands to "a^2 + 2*a*b + b^2
```

Note the quirky string output in the above example. This is because, by default, `Write` writes its arguments using the `InputForm` format, and strings in `InputForm` show their double quotes. You can override the format for each argument on an individual basis:

Subsequent `Write` commands append to the file.

```
Write[s, (a + b)^2, OutputForm[" expands to "],
        Expand[(a + b)^2]]
!!tmp
(a + b)^2" expands to "a^2 + 2*a*b + b^2
(a + b)^2 expands to a^2 + 2*a*b + b^2
```

The default format is `InputForm` because, as the next example shows, `OutputForm` is unsuitable for subsequent input whenever exponents or fractions are involved.[4]

The exponent is written on a different line of the output than everything else.

```
Write[s, OutputForm["The OutputForm of "],
         (a + b)^2,
         OutputForm[" is: "],
         OutputForm[(a + b)^2]
      ]
```

4. In version 3.0, the formats `StandardForm` and `TraditionalForm` behave as `OutputForm` unless the stream being written to is a front-end window.

```
Close[s];
!!tmp
(a + b)^2" expands to "a^2 + 2*a*b + b^2
(a + b)^2 expands to a^2 + 2*a*b + b^2
                                      2
The OutputForm of (a + b)^2 is: (a + b)
```

You can change the default output format used by a stream at the time the stream is opened. Here is an example of opening a stream with the default format set to `TeXForm`:

The contents of this file are suitable for input to the TeX typesetting system.

```
s = OpenWrite["tmp", FormatType->TeXForm];
Write[s, Sin[a/(b + c)]]
Close[s];
!!tmp
\sin ({a\over {b + c}})
```

There are quite a few options that can be set for a stream.

```
Options[OpenWrite]
{DOSTextFormat -> True, FormatType -> OutputForm,
  PageWidth -> 54, PageHeight -> 22,
  TotalWidth -> Infinity, TotalHeight -> Infinity,
  CharacterEncoding :> $CharacterEncoding,
  NumberMarks :> $NumberMarks}
```

You can see what the options values are for a particular stream by using `Options[`*`stream`*`]`. The interested reader should refer to [Wolfram 91] §2.10.3 or [Wolfram 96] §2.11.3 for a description of these options.

`WriteString` gives you even more control over the output than `Write`, since (a) you can construct strings that would not be syntactically valid *Mathematica* expressions, and (b) it does not automatically write a newline after every call. Strings are written without their surrounding quotes. Arguments that are not strings are evaluated, converted to strings, and then written.

```
a = 5; b = 7;
s = OpenWrite["tmp"];
WriteString[s, a, "+", b, "=", a + b, "\n"]
Close[s];
!!tmp
5+7=12
```

An alternative to passing an `OutputStream` object to `Write` or `WriteString` is to pass the name of a stream, e.g., `Write["tmp",` *`expr1`*`, ...]`. If there is no open stream with that name, the function will attempt to open one. The stream remains open for writing until you explicitly close it. Thus, unless you want to open a file in append mode or change any of the default output stream options, you don't need to open file streams explicitly.

The first argument to `Write` or `WriteString` also can be a list of output streams (or names of streams), which is called a *channel*. There are system-defined variables that contain the channels used for various types of output, such as `$Output` (evaluations and `Print` output), `$Messages` (`Message` output), and `$Display` (graphics output, discussed next). You can modify these variables to change where the kernel sends the output; an example of modifying `$Messages` was given in Section 9.1.1.

Exercises

1. It's a bad idea to try to use `WriteString` to write expressions in `OutputForm`. Open a stream `s`, execute the following, and explain what happened. Don't forget to `Close[s]` when you're done.

```
Clear[a, b]
WriteString[s, "The OutputForm of a/b is ",
                OutputForm[a/b], "\n"]
```

2. Write a function called `PutDelimitedForm` that essentially does what `DelimitedForm[`*`expr`*`] >>` *`filename`* does (see Exercise 12.2.2.3), but uses low-level I/O primitives. Rather than using the strategy discussed in that section for formatting the table of values, use a series of `Write` or `WriteString` statements to output the table.

   If you want to be really ambitious, you can remove the option `Delimiter` and add two new ones: `ItemDelimiter` and `RecordDelimiter`. The defaults for these should be `" "` (space) and `"\n"` (newline), respectively. This flexibility allows you to output extremely long records, for example, using `ItemDelimiter->"\n"` and `RecordDelimiter->"\n\n"`. This format is intelligible to programs such as UNIX's *awk*.

12.3.3 Display

`Display[`*`s`*`,` *`g`*`]` writes the *Mathematica* graphics object *`g`* to the stream *`s`* in PostScript format. If *`s`* refers to a notebook window, the graphic is rendered in the notebook. In general, however, you can use `Display` to write the PostScript to any stream, such as a file or pipe.

Like the other output commands, the first argument to `Display` can be a list of streams or strings. Note that if any of these strings is the name of a file that is not already open, `Display` opens the file, write the graphics data to it, *and then closes it*.

When you create a *Mathematica* graphic using any of the plotting commands, the return value from the function is a graphics object. The picture that you see on your screen is actually a *side effect* that is caused by passing the graphics object to the function specified by the `DisplayFunction` option.

```
Options[Plot, DisplayFunction]
{DisplayFunction :> $DisplayFunction}
```

```
$DisplayFunction
Display[$Display, #1] &
```

The system variable `$Display` contains a *channel* (a stream or list of streams) to which the PostScript form of graphics objects should be written. The notebook front end initializes `$Display` to `"stdout"`. Thus, in a roundabout way, the default value of the `DisplayFunction` option is `Display["stdout", #]&`.

The upshot of all this is that you can make a plotting command send the PostScript to a file instead of to the front end in any of the following three ways:

By setting the `DisplayFunction` option.

```
Plot[..., DisplayFunction->Display["filename", #]&]
```

By changing `$DisplayFunction`.

```
Block[{$DisplayFunction = Display["filename", #]&},
    Plot[...]]
```

By changing `$Display`.

```
Block[{$Display = "filename"}, Plot[...]]
```

The return value of `Display` is the same graphics object that is passed to it. This is important because all of the graphics commands that take the `DisplayFunction` option actually return the value returned by `Display`. If you override `DisplayFunction` or change the value of `$DisplayFunction`, you must ensure that the last thing your function does is to return the graphics object.

In version 3.0, `Display` takes an optional third argument that specifies a format for the graphics output. `"MPS"` (*Mathematica* PostScript) is the default, but other possible values include `"EPS"` (encapsulated PostScript), `"GIF"`, `"TIFF"`, `"XBitmap"` (X-Windows), `"MetaFile"` (Microsoft Windows), and `"PICT"` (MacOS). (See the entry for `Display` in [Wolfram 96] §A.10 for a complete list of supported formats.) So, for example, if you want to write a `GIF` version of every graphic in a *Mathematica* session to a separate file, as well as seeing each graphic on the screen, you could do something like the following:

```
n = 1;
$DisplayFunction = Function[g,
        Display[$Display, g];
        Display["GIFfile"<>ToString[n++]], g, "GIF"]]
```

Note that the return value of this function is the original graphics object, because the last thing the function does is to call `Display`.

12.4 High-Level Input

The most common input operation in *Mathematica* is to read a file of regularly formatted data into some kind of list structure. The `ReadList` function accomplishes this with a minimum of fuss.

12.4.1 Data types and templates

`ReadList` takes two arguments: a filename and an optional *template*. The template is an expression containing type keywords such as `Byte`, `Character`, `Expression`, `Number`, `Real`, `Record`, `String`, or `Word`. The template informs `ReadList` how the data are to be interpreted. For example:

```
!!intdata
967 682 130
375 963 64
525 839 941
144 969 891
206 325 234
```

This interprets the data in the file as numbers.

```
ReadList["intdata", Number]
{967, 682, 130, 375, 963, 64, 525, 839, 941, 144,
  969, 891, 206, 325, 234}
```

Now the data are interpreted as real numbers.

```
ReadList["intdata", Real]
{967., 682., 130., 375., 963., 64., 525., 839., 941.,
  144., 969., 891., 206., 325., 234.}
```

`Character` reads each character separately. `"\n"` is an ASCII newline character.

```
ReadList["intdata", Character] // InputForm
{"9", "6", "7", " ", "6", "8", "2", " ", "1", "3",
  "0", "\n", "3", "7", "5", " ", "9", "6", "3", " ",
  "6", "4", "\n", "5", "2", "5", " ", "8", "3", "9",
  " ", "9", "4", "1", "\n", "1", "4", "4", " ", "9",
  "6", "9", " ", "8", "9", "1", "\n", "2", "0", "6",
  " ", "3", "2", "5", " ", "2", "3", "4", "\n"}
```

`Byte` interprets each character in the file as an 8-bit integer.

```
ReadList["intdata", Byte]
{57, 54, 55, 32, 54, 56, 50, 32, 49, 51, 48, 13, 51,
  55, 53, 32, 57, 54, 51, 32, 54, 52, 13, 53, 50, 53,
  32, 56, 51, 57, 32, 57, 52, 49, 13, 49, 52, 52, 32,
  57, 54, 57, 32, 56, 57, 49, 13, 50, 48, 54, 32, 51,
  50, 53, 32, 50, 51, 52, 13}
```

`Word` breaks up the file into white-space-delimited strings.

```
ReadList["intdata", Word] // InputForm
{"967", "682", "130", "375", "963", "64", "525", "839",
  "941", "144", "969", "891", "206", "325", "234"}
```

`String` breaks the input only at newlines.

```
ReadList["intdata", String] // InputForm
{"967 682 130", "375 963 64", "525 839 941",
  "144 969 891", "206 325 234"}
```

`Expression` interprets each line as a *Mathematica* expression.

```
ReadList["intdata", Expression]
{85734220, 23112000, 414486975, 124326576, 15666300}
```

Note that in the last example, each line was interpreted as the product of three integers; these expressions evaluated immediately when `ReadList` returned them. You can prevent this from happening by using a template of the form `Hold[Expression]`:

```
ReadList["intdata", Hold[Expression]] // InputForm
{Hold[967*682*130], Hold[375*963*64],
  Hold[525*839*941], Hold[144*969*891],
  Hold[206*325*234]}
```

In fact, the template can be almost any *Mathematica* expression containing type keywords for its parts. Here are some examples:

`String` reads everything from the current position to the end of the line.

```
ReadList["intdata", {Number, String}] // InputForm
{{967, " 682 130"}, {375, " 963 64"},
  {525, " 839 941"}, {144, " 969 891"},
  {206, " 325 234"}}
```

The template argument is evaluated before `ReadList` is called.

```
ReadList["intdata", Table[Number, {5}]]
{{967, 682, 130, 375, 963}, {64, 525, 839, 941, 144},
  {969, 891, 206, 325, 234}}
```

The possibilities are limitless.

```
ReadList["intdata",
         {f[Number, HoldForm[Number + Number]]}]
{{f[967, 682 + 130]}, {f[375, 963 + 64]},
  {f[525, 839 + 941]}, {f[144, 969 + 891]},
  {f[206, 325 + 234]}}
```

`ReadList["`*`filename`*`", `*`template`*`, `*`n`*`]` reads at most *n* objects that match the specified template from a file.

```
ReadList["intdata", Number, 5]
{967, 682, 130, 375, 963}
```

Note, however, that every call to `ReadList["`*`filename`*`", ...]` starts reading at the beginning of the file. If the first argument to `ReadList` is an *input stream* rather than a character string, the position of the last read is remembered, so that different parts of the file can be interpreted differently. This is particularly useful when the first few lines of a file are descriptive data, column labels, etc. We'll cover input streams in Section 12.5.

12.4.2 Reading numbers

`ReadList` is terrific when the input data consist of numerical values separated by white-space. The simplest way to read such data is to read each number separately, as illustrated earlier:

```
ReadList["intdata", Number]
{967, 682, 130, 375, 963, 64, 525, 839, 941, 144,
  969, 891, 206, 325, 234}
```

You can specify that each line should be a separate sublist (thus maintaining the tabular form of the input) using the `RecordLists` option.

```
ReadList["intdata", Number, RecordLists->True]
{{967, 682, 130}, {375, 963, 64}, {525, 839, 941},
  {144, 969, 891}, {206, 325, 234}}
```

Alternatively, you can group the numbers any way you wish by using template expressions.

There weren't enough data to fill the final record, so `ReadList` returned `EndOfFile`.

```
ReadList["intdata", {Number, Number}]
{{967, 682}, {130, 375}, {963, 64}, {525, 839},
  {941, 144}, {969, 891}, {206, 325}, {234, EndOfFile}}
```

The `Number` format will adjust to integers or decimal numbers automatically. Specifying `Real` instead of `Number` will cause all numerical inputs to be treated as approximate, even if they contain no decimal point (see the example in the previous section). Both `Real` and `Number` formats can interpret Fortran-style scientific notation.

This file was created in Section 12.2.2.

```
!!edatafile
12. 162754.7914190039
13. 442413.3920089206
14. 1.202604284164777e6
15. 3.269017372472111e6
16. 8.88611052050787e6
17. 2.41549527535753e7

ReadList["edatafile", Number]
{12., 162754.7914190039, 13., 442413.3920089206, 14.,
  1.2026 10^6, 15., 3.26902 10^6, 16., 8.88611 10^6,
  17., 2.4155 10^7}
```

12.4.3 Reading strings

There are three different ways to read strings from a file, using the `String`, `Word`, and `Record` type keywords. `String` is the simplest of these:

```
!!worddata
The quick, brown, fox jumps
over the lazy dog.
```

Objects of type `String` are delimited by newlines.

```
ReadList["worddata", String]     // InputForm
{"The quick, brown, fox jumps", "over the lazy dog."}
```

If you want more control over how the input is broken up, you can read objects of type `Record` and specify a list of `RecordSeparators` (which defaults to newline).

```
ReadList["worddata", Record, RecordSeparators->{","}
]   // InputForm
{"The quick", " brown",
  " fox jumps\nover the lazy dog.\n"}
```

Record separators do not have to be single characters.

```
ReadList["worddata", Record,
        RecordSeparators->{"fox", "the"}
]   //  InputForm
{"The quick, brown, ", " jumps\nover ",
  " lazy dog.\n"}
```

Alternatively, you can tell `ReadList` that you want it to break up the input into `Words`, which by default are white-space-delimited character strings.

```
ReadList["worddata", Word]       // InputForm
{"The", "quick,", "brown,", "fox", "jumps",
   "over", "the", "lazy", "dog."}
```

You can specify a list of `Word` delimiters using the `WordSeparators` option (the default is space and tab). In that case, what is considered to be a `Word` by `ReadList` may not be a "word" in the conventional sense.

```
ReadList["worddata", Word, WordSeparators->{","}
]   // InputForm
{"The quick", " brown", " fox jumps",
   "over the lazy dog."}
```

`Words` are broken at both word and record separators, which is why words do not normally cross line boundaries. You can change this behavior by specifying a different value for the `RecordSeparators` option.

The fifth and eighth "words" span line boundaries.

```
ReadList["worddata", Word, RecordSeparators->{}
]   // InputForm
{"The", "quick,", "brown,", "fox",
   "jumps\nover", "the", "lazy", "dog.\n"}
```

12.4.4 Application: Reading comma-delimited data

Okay, now for the bad news: `ReadList` gives only minimal support for this common case. The file *csv* contains the data that we used in our export example, except that it is in "comma-separated values" format.

```
!!csv
12.,162754.7914190039
13.,442413.3920089206
14.,1.202604284164777e6
15.,3.269017372472111e6
16.,8.88611052050787e6
17.,2.41549527535753e7
```

If you simply want to read each number into a flat list, there is no problem: Starting with version 2.2 you can specify custom record separators when reading numbers.

```
ReadList["csv", Number, RecordSeparators->{","}]
{12., 162754.7914190039, 13., 442413.3920089206, 14.,
  1.2026 10^6, 15., 3.26902 10^6, 16., 8.88611 10^6,
  17., 2.4155 10^7}
```

However, you cannot read structured data in this way.

`ReadList` got stuck when it hit the first comma, because it was looking for the second number.[5]

```
ReadList["csv", {Number, Number},
        RecordSeparators->{","}]
Read::readn:
   Syntax error reading a real number from csv.
{{12., $Failed}}
```

One way around this problem is to impose a structure on the list of numbers *after* reading it.

```
Partition[%%, 2]
{{12., 162754.7914190039}, {13., 442413.3920089206},
  {14., 1.2026 10^6}, {15., 3.26902 10^6},
  {16., 8.88611 10^6}, {17., 2.4155 10^7}}
```

Note two problems with this approach: First, the structure must be regular, i.e., there must be the same number of data items on each line of the file. The second, more serious problem is that some programs (notably spreadsheets) will output multiple delimiters in a row when there are empty cells in the range of cells being exported:

```
!!multicomma
1,,24,13,,8
2,12,,,,16
3,7,17,,20,
```

The approach outlined above is hopeless here. The only alternative seems to be to read the file as `Words`, specifying a comma as the delimiter — but even this is not straightforward.

By default, `ReadList` will ignore multiple delimiters in a row. The original structure of the data has been lost.

```
ReadList["multicomma", Word, WordSeparators->",",
    RecordLists->True] // InputForm
{{"1", "24", "13", "8"}, {"2", "12", "16"},
  {"3", "7", "17", "20"}}
```

5. Note that in version 2.2, `ReadList` erroneously returns `{{12., EndOfFile}}` instead of `{{12., $Failed}}`.

The solution is to specify `NullWords->True`. Missing items are now represented explicitly by empty strings.

```
ReadList["multicomma", Word, WordSeparators->{","},
    RecordLists->True, NullWords->True] // InputForm
{{"1", "", "24", "13", "", "8"},
  {"2", "12", "", "", "", "16"},
  {"3", "7", "17", "", "20"}}
```

Finally, convert the strings to integers. You can easily replace the `Null`s with `0`s if desired.

```
Map[ToExpression, %, {2}]
{{1, Null, 24, 13, Null, 8},
  {2, 12, Null, Null, Null, 16}, {3, 7, 17, Null, 20}}
```

There's still a problem with this approach, which the following exercise will fix.

Exercises

1. Try reading the *csv* data using the approach developed in this section. Be sure to look at the internal form of the result. Can you fix this problem? (*Hint:* A normal pattern won't do the job. You need to solve the problem *before* converting the strings to expressions. Try using `StringReplace`.) Write a function that wraps up all of these steps. Be sure to handle empty values by replacing them with `0`s.
2. Override the definition of `ReadList` so that `ReadList["filename", CSV]` calls the function you just defined. You should do this using an *upvalue* for the symbol `CSV` rather than a *downvalue* for `ReadList` (see Section 6.5.2, "Upvalues).

12.5 Low-Level Input

Corresponding to the low-level output primitives, there are low-level input primitives that can be used to read just about any kind of data, if you are willing to work hard enough at it.

12.5.1 OpenRead

`OpenRead` is analogous to `OpenWrite`. It returns an `InputStream` rather than an `OutputStream`:

```
s = OpenRead["tmp"]
InputStream[tmp, 21]
```

As in the case of output streams, you should close an input stream after you are done with it.

```
Close[s];
```

12.5.2 Reading from a stream

`Read` allows you to read an object from a stream, where "object" is any of the types supported by `ReadList`.

```
s = OpenRead["edatafile"];
```

Here we read a number and a word.

```
Read[s, {Number, Word}] // InputForm
{12., "162754.7914190039"}
```

Next, three strings. The first string read is the end of the current line.

```
Read[s, Table[String, {3}]] // InputForm
{"", "13. 442413.3920089206",
   "14. 1.202604284164777e6"}
```

`Read` keeps track of your position within the stream. You can get the current position using the `StreamPosition` function, and you can set the position using `SetStreamPosition`.

Save the current position.

```
p = StreamPosition[s];
```

Read four numbers.

```
Read[s, Table[Number, {4}]]
{15., 3.26902 10^6, 16., 8.88611 10^6}
```

Return to the saved position.

```
SetStreamPosition[s, p];
```

Read four strings instead. Note the `EndOfFile` indication.

```
Read[s, Table[String, {4}]] // InputForm
{"15. 3.269017372472111e6", "16. 8.88611052050787e6",
   "17. 2.41549527535753e7", EndOfFile}
```

At any point during the course of reading from an input stream, you can use `ReadList` to read everything from the current position to the end of the stream.

```
SetStreamPosition[s, p];
ReadList[s, String] // InputForm
{"15. 3.269017372472111e6", "16. 8.88611052050787e6",
   "17. 2.41549527535753e7"}
```

Alternatively, you can use `ReadList[s, template, n]` to read at most *n* objects of the specified template from the stream `s`. Streams allow you to mix calls to `Read` and `ReadList` without losing your position.

Don't forget to close the stream.

```
Close[s];
```

If you try to read an incorrect type of object, `Read` will return the special symbol `$Failed`.[6]

This file contains comma-separated numbers.

```
s = OpenRead["csv"];
Read[s, Number]
12.
```

The next character in the file is a comma.

```
Read[s, Number]
Read::readn:
   Syntax error reading a real number from csv.
$Failed
```

You can use `Skip` to skip over items that you don't want to read.

```
Skip[s, Character]

Read[s, Number]
162754.7914190039

Close[s];
```

`Skip` takes an optional third argument that specifies how many items to skip. A particularly useful application of this is to use `Skip[`*`s`*`, String, `*`n`*`]` to skip *n* lines of the stream *s*.

Exercise

1. Use the low-level input functions to write a routine that reads a file of comma-separated values.

12.6 Additional Resources

[Shaw & Tigg 94] contains many practical examples of reading data from files.

6. Note that in versions 2.2 and earlier, `Read` erroneously returns `EndOfFile` under these circumstances. This does not prevent you from continuing to read the file, however.

13

Debugging

Most of the time the interactive nature of *Mathematica* makes debugging easy. There are occasions, however, on which executing a function step by step in search of a bug is unacceptably tedious. In earlier chapters we saw a few uses of the `Trace` command to enable postmortem examination of the evaluation of *Mathematica* expressions, but until now we have ignored the details of this technique. `Trace` has many options that we will explore in Section 13.1.

There are some additional, more interactive, debugging tools in *Mathematica* that few users know about and even fewer use. These tools, which are the topic of Section 13.2, are quite unlike the interactive debuggers that users of compiled languages are familiar with.

13.1 Tracing Evaluations

The `Trace` function (and its relatives) can be used to trace the evaluation of an expression. We have already seen uses of `Trace` scattered throughout the earlier chapters. In this section we present a comprehensive discussion of `Trace`.

All of the `Trace` outputs in this chapter were generated using *Mathematica* version 3.0, which in certain instances gives different output than version 2.2. Mostly, this difference consists of certain trivial evaluations that appear in version 2.2 traces that do not appear in version 3.0 traces. Less commonly, the version 3.0 traces contain expressions that version 2.2 omits.

13.1.1 Trace basics

`Trace[`*`expr`*`]` returns a list of all intermediate forms encountered during the evaluation of *`expr`*.

```
Trace[2 - 3 * 4]
{{-(3 4), -12}, 2 - 12, -10}
```

This trace shows that `-3 * 4` evaluated to `-12`. Then `2 - 12` was evaluated, yielding the final answer, `-10`.

The list returned by `Trace` is nested so that the evaluation of every subexpression appears within its own sublist. Each such sublist is called an *evaluation chain*. Thus `{-(3 4), -12}` is the complete evaluation of `Times`; this list is nested within the outer list because its result is one of the arguments to `Plus`. The outer pair of list braces corresponds to everything within the `Plus`.

Note that every expression in the return value of `Trace` is wrapped in `HoldForm`, which keeps its argument from evaluating (but is invisible when formatted in `OutputForm`).

```
InputForm[%]
{{HoldForm[-(3*4)], HoldForm[-12]}, HoldForm[2 - 12],
  HoldForm[-10]}
```

Here is an example involving a head with an `OwnValue`. The `Trace` shows that the head is evaluated first, and then the arguments in the order in which they appear. The evaluation chain for the head is placed in its own sublist, just like the evaluation chain for each argument:[1]

```
f = #1===#2 &;
x = 7; y = 7;
Trace[f[x, y]]
{{f, #1 === #2 & }, {x, 7}, {y, 7},
  (#1 === #2 & )[7, 7], 7 === 7, True}
```

Here's an example of `Hold-` attributes.

```
Attributes[g] = HoldFirst;
g[a_, b_] := a === b

Trace[g[x, y]]
{{y, 7}, g[x, 7], x === 7, {x, 7}, 7 === 7, True}
```

Note that even though `x` is passed into `g` unevaluated, it gets evaluated when it is passed to `SameQ` ("`===`"). To prevent that, use `Unevaluated`:

```
Attributes[h] = HoldFirst;
h[a_, b_] := Unevaluated[a] === b
```

1. This actually makes a lot of sense, since the head is a part (in the technical sense) of a normal expression just like any other part. The head just happens to be part 0.

```
Trace[h[x, y]]
{{y, 7}, h[x, 7], Unevaluated[x] === 7, x === 7,
  False}
```

Here is an example of how tracing can give us some insight into the operation of the kernel's main evaluation loop. The `Orderless` attribute tells the kernel that a function is commutative. The kernel sorts the arguments to an orderless function into some well-defined canonical order before calling the function; this simplifies the logic of many algebraic simplification routines. For example, the terms in a polynomial are ordered by their exponents, making operations such as comparison for equality much simpler.

```
Trace[Expand[(1 + z)^2] == z^2 + 2 z + 1]
{{Expand[(1 + z)^2], 1 + 2 z + z^2},
  {z^2 + 2 z + 1, 1 + 2 z + z^2},
  1 + 2 z + z^2 == 1 + 2 z + z^2, True}
```

However, attributes like `Orderless` (this include `Flat`) have another effect. If a rule does not match after the argument ordering is done, the pattern matcher will try other argument orders to see if the rule can be made to match.[2] This behavior is exploited by the following function that merges two sorted lists:

```
SetAttributes[merge, Orderless]
merge[a:{a1_, arest___}, b:{b1_, brest___}] :=
    Prepend[merge[{arest}, b], a1] /; a1 <= b1
merge[{}, b_List] := b
```

Giving `merge` the `Orderless` attribute eliminates the need for "mirror image" rules that are identical to the two already given but with the order of the arguments reversed (cf. the implementation of `merge` in Section 6.3.6).

Here is a simple trace of `merge[{1, 3}, {2}]`. Note that the arguments are reversed immediately because the canonical order dictates that shorter lists precede longer ones. However, after the conditional test (`a1 <= b1`) fails, the rule is tried again with the arguments in the other (original) order, resulting in a recursive call to `merge[{3}, {2}]`:

```
Trace[merge[{1, 3}, {2}]]
{merge[{1, 3}, {2}], merge[{2}, {1, 3}],
  {{2 <= 1, False}, RuleCondition[Prepend[merge[{},
      {1, 3}], 2], False], Fail},
  {{1 <= 2, True}, RuleCondition[Prepend[merge[{3},
      {2}], 1], True], Prepend[merge[{3}, {2}], 1]},
  Prepend[merge[{3}, {2}], 1],
  {merge[{3}, {2}], merge[{2}, {3}],
   {{2 <= 3, True}, RuleCondition[Prepend[merge[{},
```

2. In the case of `Flat`, the pattern matcher tries different pattern *groupings*.

```
    {3}], 2], True], Prepend[merge[{}, {3}], 2]},
 Prepend[merge[{}, {3}], 2], {merge[{}, {3}], {3}},
 Prepend[{3}, 2], {2, 3}}, Prepend[{2, 3}, 1],
{1, 2, 3}}
```

`merge[{3},{2}]` is then reordered to `merge[{2},{3}]` before checking the condition, which turns out to be the right thing to do.

The output of `Trace` can be quite voluminous. If you are using the notebook front end, it may be helpful to keep the definition of the function being traced and the actual traces in separate windows; this eliminates a lot of scrolling up and down.

13.1.2 Restricting the trace

We saw in the previous section that the output of `Trace` can be large even when tracing trivial computations; it typically is *enormous* when tracing nontrivial ones. For example, it would be quite difficult to understand the trace of the `merge` function on any realistically sized inputs. To alleviate this problem, `Trace` takes a second argument that can be used to restrict the information in the trace. There are two forms for this argument.

One form of the argument is called a *tag*. A tag is a symbol with which a particular transformation rule is associated. It is usually the head of an expression, but it also may be a symbol at level 1 for which an `UpValue` has been defined (refer to Sections 6.5.2 and 7.1.1).

For example, here we trace all subexpressions in the evaluation of `merge[{1, 3, 5}, {2, 4}]` that are tagged with the symbol `merge`:

This is much more digestible than a full trace.

```
Trace[merge[{1, 3, 5}, {2, 4}], merge]
{merge[{1, 3, 5}, {2, 4}], merge[{2, 4}, {1, 3, 5}],
  Prepend[merge[{3, 5}, {2, 4}], 1],
  {merge[{3, 5}, {2, 4}], merge[{2, 4}, {3, 5}],
   Prepend[merge[{4}, {3, 5}], 2],
   {merge[{4}, {3, 5}], Prepend[merge[{5}, {4}], 3],
    {merge[{5}, {4}], merge[{4}, {5}],
     Prepend[merge[{}, {5}], 4],
     {merge[{}, {5}], {5}}}}}}
```

The reordering of the arguments is quite explicit here. Also note that it is quite easy to see the recursive structure of the computation.

When a tag is used as the second argument to `Trace`, `Trace` includes not only the expressions that match rules associated with this tag, but also the results of evaluating those rules (in the case of the tag `merge`, a call to `Prepend`). We can restrict the trace even further by specifying a pattern as the second argument to `Trace`. `Trace` will then include only those intermediate expressions that match the pattern in the returned trace.

```
Trace[merge[{1, 3, 5}, {2, 4}], merge[_, _]]
{merge[{1, 3, 5}, {2, 4}], merge[{2, 4}, {1, 3, 5}],
  {merge[{3, 5}, {2, 4}], merge[{2, 4}, {3, 5}],
   {merge[{4}, {3, 5}],
    {merge[{5}, {4}], merge[{4}, {5}],
     {merge[{}, {5}]}}}}}
```

Exercise

1. Reproduce the last trace in this section by creating a full trace and filtering it with `Cases`. *Hint:* Use a level specification (Section 5.1.3) of `Infinity`.

13.1.3 Looking forward or backward

Note that the traces in the previous section did not reveal what the eventual outcome of each traced expression was. We can obtain this information using the `TraceForward` option:

```
Trace[merge[{1, 3, 5}, {2, 4}], merge[_, _],
      TraceForward->True]
{merge[{1, 3, 5}, {2, 4}], merge[{2, 4}, {1, 3, 5}],
  {merge[{3, 5}, {2, 4}], merge[{2, 4}, {3, 5}],
   {merge[{4}, {3, 5}],
    {merge[{5}, {4}], merge[{4}, {5}],
     {merge[{}, {5}], {5}}, {4, 5}}, {3, 4, 5}},
   {2, 3, 4, 5}}, {1, 2, 3, 4, 5}}
```

`TraceForward->True` shows the final expression in each evaluation chain that is traced. Now we can see, for example, that `merge[{}, {5}]` evaluated to `{5}` and that `merge[{5}, {4}]` evaluated to `{4, 5}`.

A similar option, `TraceForward->All`, shows *all* intermediate results in the evaluation chain from the traced form until the end of the chain. Because of the volume of output this produces, the next example restricts the traced forms to those calls to `merge` that have `{5}` as their second argument.

```
Trace[merge[{1, 3, 5}, {2, 4}], merge[_, {5}],
      TraceForward->All]
{{{{merge[{4}, {5}], Prepend[merge[{}, {5}], 4],
     {merge[{}, {5}], {5}}, Prepend[{5}, 4], {4, 5}}}}
  }
```

Often, however, what is desired is to find out where a particular subexpression came *from*. For example, suppose we want to see what evaluation gave rise to the intermediate result `{3, 4, 5}`. This can be done with the `TraceBackward` option.

```
Trace[merge[{1, 3, 5}, {2, 4}], {3, 4, 5},
      TraceBackward->True]
{{{merge[{4}, {3, 5}], {3, 4, 5}}}}
```

`TraceBackward->True` shows the first expression in each evaluation chain that matches the traced form. In a fashion analogous to the operation of `TraceForward`, `TraceBackward->All` shows all intermediate results between the beginning of the chain and the traced form.

```
Trace[merge[{1, 3, 5}, {2, 4}], {3, 4, 5},
      TraceBackward->All]
{{{merge[{4}, {3, 5}], Prepend[merge[{5}, {4}], 3],
    Prepend[{4, 5}, 3], {3, 4, 5}}}}
```

Sometimes the information you need is contained not in the evaluation chain for the traced form, but in the chain that contains that chain. An example of this occurs when the form being traced is generated by a side effect. Here is a ridiculously simple example of some code that causes an error message to be generated:

```
isZero[n_] :=
    If[n = 0, True, False]

isZero[3]
Set::setraw: Cannot assign to raw object 3.
If[0, True, False]
```

To find the bug, we attempt to find out what caused the call to the `Message` function:

The expression that triggered the call to `Message` is not in this evaluation chain.

```
Trace[isZero[3], Message,
      TraceForward->All, TraceBackward->All]
Set::setraw: Cannot assign to raw object 3.
{{{Message[Set::setraw, 3], Null}}}
```

It is in the chain above this one.

```
Trace[isZero[3], Message, TraceAbove->True]
Set::setraw: Cannot assign to raw object 3.
{isZero[3], {3 = 0, {Message[Set::setraw, 3], Null},
   0}, If[0, True, False]}
```

By tracing the chain above the one containing `Message`, we see that the message was generated because the "test" inside the `If` actually was an assignment.

13.1.4 TraceOn, TraceOff

The `TraceOn` and `TraceOff` options control which parts of a computation are traced. For example, suppose we already had debugged the `merge` function, and now we want to debug `mergesort` (Section 5.4.2):

```
mergesort[s:{_} | s:{}] := s
mergesort[s_List] :=
    With[{half = Quotient[Length[s], 2]},
        merge[mergesort[Take[s, half]],
              mergesort[Drop[s, half]]]
    ]
```

There is no reason to trace *within* calls to `merge`, since we presume that we are confident that `merge` works. We can use the `TraceOff` option to prevent calls to `merge` from being traced during the tracing of `mergesort`.

This call traces everything *except* calls to `merge`, `Quotient`, and `Length`.

```
Trace[mergesort[{5, 3, 2}],
      TraceOff->merge[__] | Quotient[__] | Length[_]]
{mergesort[{5, 3, 2}],
  With[{half$ = Quotient[Length[{5, 3, 2}], 2]},
   merge[mergesort[Take[{5, 3, 2}, half$]],
    mergesort[Drop[{5, 3, 2}, half$]]]],
  {{Length[{5, 3, 2}], 3}, Quotient[3, 2], 1},
  merge[mergesort[Take[{5, 3, 2}, 1]],
   mergesort[Drop[{5, 3, 2}, 1]]], {2, 3, 5}}
```

Conversely, `TraceOn` can be used to turn tracing *on* only within certain forms.

This call traces *only* calls to `merge`, and only those for which both arguments are single-element lists.

```
Trace[mergesort[{5, 3, 2}],
      TraceOn->merge[{_}, {_}]]
{{merge[{3}, {2}], merge[{2}, {3}],
   {{2 <= 3, True}, RuleCondition[Prepend[merge[{},
       {3}], 2], True], Prepend[merge[{}, {3}], 2]},
   Prepend[merge[{}, {3}], 2], {merge[{}, {3}], {3}},
   Prepend[{3}, 2], {2, 3}},
  {merge[{3}, {5}], {{3 <= 5, True},
    RuleCondition[Prepend[merge[{}, {5}], 3], True],
    Prepend[merge[{}, {5}], 3]},
   Prepend[merge[{}, {5}], 3], {merge[{}, {5}], {5}},
   Prepend[{5}, 3], {3, 5}}}
```

13.1.5 TraceDepth

`TraceDepth` is another option that limits the amount of information present in a trace. It simply eliminates all information below a certain depth in the trace.

By setting `TraceDepth` to 1 in this example, we do not see a trace of any of the lower-level recursive calls.

```
Trace[mergesort[{5, 3, 2}], TraceDepth->1]
{mergesort[{5, 3, 2}],
  With[{half$ = Quotient[Length[{5, 3, 2}], 2]},
   merge[mergesort[Take[{5, 3, 2}, half$]],
    mergesort[Drop[{5, 3, 2}, half$]]]],
  merge[mergesort[Take[{5, 3, 2}, 1]],
   mergesort[Drop[{5, 3, 2}, 1]]],
```

```
merge[{5}, {2, 3}], Prepend[merge[{3}, {5}], 2],
Prepend[{3, 5}, 2], {2, 3, 5}}
```

13.1.6 MatchLocalNames

The names of all local symbols declared in a `Module` command are made unique by appending a `$` followed by the current value of `$ModuleNumber` (Section 4.1.4). By default, `Trace` will match names like `x$`*nn* to the pattern `x_`, which is very convenient. Consider the following recursive function that calculates the length of a list:

```
len[{a_, b___}] := Module[{s = {b}}, 1 + len[s]]
len[{}] = 0;

len[{1, 2, 3}]
3
```

We can trace all assignments to the local symbol `s` using the following command:

The name of the local symbol `s` is different in each recursive call, but `Trace` matches them all to the pattern `s_`.

```
Trace[len[{1, 2, 3}], s = _]
{{s$4 = {2, 3}}, {{{s$5 = {3}}, {{{s$6 = {}}}}}}}
```

In certain cases this behavior may not be desirable: for example, when there are multiple `Modules` having the same local variable name, or if there is a global variable of the same name that is the true subject of interest. In such cases you can disable this feature with the `MatchLocalNames->False` option setting.

In case you want to trace only *certain* local variables with a given name, you can, with a bit of effort, figure out what the unique variable name will be (using `$ModuleNumber`) and use that in the trace pattern.

```
$ModuleNumber
7
```

The nesting of the result reflects the structure of the entire computation.

```
Trace[len[{1, 2, 3, 4, 5, 6}], s$10 = _]
{{{{{{{{s$10 = {5, 6}}}}}}}}}
```

13.1.7 TraceOriginal

`TraceOriginal->True` is an option to `Trace` and related functions that allows you to see expressions before their parts have been evaluated. Compare the following two examples to see the effect of this option:

```
Trace[Cos[.7] + Sqrt[2.5], Plus[__]]
{0.764842 + 1.58114}
```

```
Trace[Cos[.7] + Sqrt[2.5], Plus[__],
      TraceOriginal->True]
{Cos[0.7] + Sqrt[2.5], 0.764842 + 1.58114}
```

13.2 Interactive Debugging

The tracing techniques that we examined in the last section all have one thing in common: All they provide is postmortem information. Debugging using them is a process of successive refinement, as we repeatedly trace a computation with different options, trying to "zero in" on the problem. Sometimes this is not practical — a computation that fails to terminate is an obvious example — and in such cases the only alternative is to interact with the computation as it executes. This mode of debugging probably is familiar to programmers of other programming languages.

13.2.1 TracePrint

`TracePrint` prints the trace, one expression per line, as the computation proceeds. It is interactive in the sense that it can be used to monitor a badly behaving computation and then abort the computation from the keyboard as soon as it's clear what is going wrong.

Since the output of `TracePrint` is so verbose, we return to tracing a very simple example.

Note that the expressions are indented according to their level in the trace.

```
TracePrint[2 - 3 * 4]
 2 - 3 4
  Plus
  2
  -(3 4)
   Times
   -1
   3
   4
  -12
 2 - 12
 -10
-10
```

As usual, you can limit the trace using a tag or pattern as a second argument.

13.2.2 TraceDialog

The most powerful interactive debugging command is `TraceDialog`. `TraceDialog[`*`expr`*`, `*`form`*`, `*`options`*`]` enters a *dialog* each time *`form`* occurs during the evaluation of *`expr`*. A dialog is a subsidiary *Mathematica* session in the midst of some other evaluation during which your interactive commands have the full attention of the kernel. A dialog can be initiated at any time by calling the function `Dialog`. While inside a

dialog, the in/out prompts (shown below on the left) are different to remind you of this fact.

```
In[1]:=     3 + Dialog[2]
Out[2]=     2
```

Note that the line number of the output is not 1, but rather 2; this is because the kernel has implicitly assigned the argument of `Dialog` to `In[2]`, and the result of evaluating that argument to `Out[2]`. Thus, you can refer to that value using `%`:

```
(Dialog) In[3]:=     %^2
(Dialog) Out[3]=     4
```

You can perform any calculation you wish within the dialog (including additional calls to `Dialog`). To exit from a `Dialog`, use the `Return` function. By default, `Dialog[`*expr*`]` returns *expr*, but if you provide an argument to `Return`, that argument becomes the value of the call to `Dialog`. For example, the following `Return` statement causes the call to `Dialog[2]` to evaluate to 4:

```
(Dialog) In[4]:=     Return[%]
```

Now a strange thing happens if you are using the notebook interface: The output corresponding to the above input does not print here; it prints higher up in the notebook, immediately below the original call to `Dialog`. Furthermore, the line number is reset so that the output created logically corresponds to the original input. What you will see in the notebook is something like this:

```
In[1]:=     3 + Dialog[2]
Out[2]=     2
Out[1]=     7
```

In the remainder of this section we will not show line numbers, but you should be aware that line numbering and output placement in the notebook will be nonmonotonic. You may find it is easier to follow what is going on by doing the dialog calculations in a different notebook than the original call to `TraceDialog`.

We'll demonstrate the use of `TraceDialog` on the following function, which iteratively calculates a rational approximation to the square root of its argument.

```
iterSqrt[x_] :=
Module[{i, j, xn = 1},
    While[Abs[xn^2 - x] > 10^-6,
        i = x / xn;
        j = xn + i;
        xn = j / 2
    ];
    xn
]
```

```
iterSqrt[2]
```

$$\frac{665857}{470832}$$

```
N[%]
1.41421
```

As discussed in Section 13.1.6, `Trace` will match the pattern `xn_` to all local variables constructed from that name.

```
Trace[iterSqrt[2], xn = _]
```

$$\{\{xn\$14 = 1\}, \{\{xn\$14 = \frac{3}{2}\}\}, \{\{xn\$14 = \frac{17}{12}\}\},$$
$$\{\{xn\$14 = \frac{577}{408}\}\}, \{\{xn\$14 = \frac{665857}{470832}\}\}\}$$

Suppose now that we want to stop each time that `xn` is about to be assigned a value, presumably so that we could inspect or alter the state of the computation. This is done with `TraceDialog`:

```
TraceDialog[iterSqrt[2], xn = _]
TraceDialog::dgbgn:
   Entering Dialog; use Return[ ] to exit.
xn$15 = 1
```

The expression matching the pattern is passed into the dialog, wrapped in `HoldForm` to prevent evaluation.

```
FullForm[%]
HoldForm[Set[xn$15, 1]]
```

We exit the dialog by evaluating `Return[]`.

```
Return[]
TraceDialog::dgend: Exiting Dialog.
```

The dialog is entered again when the assignment to `xn` at the bottom of the loop is encountered.

```
TraceDialog::dgbgn:
   Entering Dialog; use Return[ ] to exit.
```

$$xn\$15 = \frac{j\$15}{2}$$

Note that the arguments to `Set` have not yet been evaluated. This would allow us to change the expression on the right-hand side and thus affect the assignment to `xn` if we so desired. We can see what the value will be by evaluating the right-hand side manually:

```
j$15/2
```

$$\frac{3}{2}$$

Resume evaluation again. A new dialog is entered after the right-hand side of the set is evaluated.

```
Return[]
TraceDialog::dgend: Exiting Dialog.
TraceDialog::dgbgn:
   Entering Dialog; use Return[ ] to exit.
```

$$\text{xn\$15} = \frac{3}{2}$$

Seeing every assignment statement twice is rather tedious. We can change this behavior by changing the pattern that `Trace` is searching for, even in the midst of a traced evaluation! This pattern is stored in the global variable `$TracePattern`. Note that in general we can't look at the value of `$TracePattern` directly, because it might immediately evaluate to something else. But we can look at it indirectly, by inspecting its `OwnValues`:

The `RuleDelayed` prevents `xn = _` from being evaluated.

```
OwnValues[$TracePattern]
{HoldPattern[$TracePattern] :> (xn = _)}
```

Suppose that we want to see only those assignments in which the right-hand side is already evaluated. Getting the pattern right is slightly tricky.

Set `$TracePattern` to a pattern that matches only numbers.

```
$TracePattern := xn = _?NumberQ
```

Why didn't it work?

```
Return[]
TraceDialog::dgend: Exiting Dialog.
TraceDialog::dgbgn:
   Entering Dialog; use Return[ ] to exit.
```

$$\text{xn\$15} = \frac{\text{j\$15}}{2}$$

The problem is that `j$15` evaluates when it is passed to `NumberQ`.

```
NumberQ[j$15/2]
True
```

The solution is to use the usual trick of wrapping the argument in `Unevaluated`.

```
$TracePattern := xn = y_ /; NumberQ[Unevaluated[y]]
```

This doesn't prove anything since this is the very next assignment to `xn`.

```
Return[]
TraceDialog::dgend: Exiting Dialog.
TraceDialog::dgbgn:
   Entering Dialog; use Return[ ] to exit.
```

$$\text{xn\$15} = \frac{17}{12}$$

Let's try one more time. The intermediate form of the assignment has been skipped, exactly as promised!

```
Return[]
TraceDialog::dgend: Exiting Dialog.
TraceDialog::dgbgn:
   Entering Dialog; use Return[ ] to exit.
xn$15 = 577/408
```

We can modify any values that we desire while in the dialog. Let's go to the next step after the assignment to xn, and then set xn to something else.

This is about the closest thing to "single-stepping" a computation that we can get in *Mathematica*.

```
$TracePattern = _;
```

The next two steps are a continuation of the evaluation of the Set command.

```
Return[]
TraceDialog::dgend: Exiting Dialog.
TraceDialog::dgbgn:
   Entering Dialog; use Return[ ] to exit.
577
408

Return[]
TraceDialog::dgend: Exiting Dialog.
TraceDialog::dgbgn:
   Entering Dialog; use Return[ ] to exit.
577
408
```

Eventually, TraceDialog stops at the next "statement" in the function.

```
Return[]
TraceDialog::dgend: Exiting Dialog.
TraceDialog::dgbgn:
   Entering Dialog; use Return[ ] to exit.
Abs[xn$15^2 - 2] > 10^-6
```

Now let's change xn to a machine-precision number. This should cause all subsequent calculations to be done using machine-precision arithmetic.

```
xn$15 = 1.75
1.75
```

Reset $TracePattern.

```
$TracePattern := xn = y_ /; NumberQ[Unevaluated[y]]
```

We omit the TraceDialog messages for brevity.

```
Return[]
xn$15 = 1.44643

Return[]
xn$15 = 1.41457
```

```
Return[]
xn$15 = 1.41421
```

The computation seems to be proceeding perfectly well; let's get out of `TraceDialog` mode. The easiest way to do this is to set `$TracePattern` to something that will never be encountered.

The computation finishes without further interruption.

```
$TracePattern = "a blue moon";

Return[]
TraceDialog::dgend: Exiting Dialog.
1.41421
```

Note that although you can return a value from a `Dialog`, that value is ignored by `TraceDialog`. In particular, there's no direct way to abort a computation in progress. However, you can usually do it indirectly:

In this example we trace assignments to `j`.

```
TraceDialog[iterSqrt[2], j = _]
TraceDialog::dgbgn:
   Entering Dialog; use Return[ ] to exit.
j$16 = xn$16 + i$16
```

The computation aborts as soon as `xn` is evaluated.

```
xn$16 := Abort[]
Return[]
TraceDialog::dgend: Exiting Dialog.
$Aborted
```

You can specify your own function to be called each time the trace stops by using the `TraceAction` option. The default value for `TraceAction` is shown below.

```
Options[TraceDialog, TraceAction]
{TraceAction ->
   (StackInhibit[Message[TraceDialog::dgbgn];
       Dialog[#1]; Message[TraceDialog::dgend]] & )}
```

As you can see, the `TraceDialog` messages are generated here; if you find them particularly bothersome, you can eliminate them by specifying `TraceAction->StackInhibit[Dialog[#]]&`. The `StackInhibit` function will be discussed in the next section.

13.2.3 The evaluation stack

Mathematica records computations in progress on the *evaluation stack*, which can be viewed at any time using the `Stack` function. It is quite useful in combination with `TraceDialog`, as it shows how the computation came to be at its current point.

```
TraceDialog[iterSqrt[2], j = _]
TraceDialog::dgbgn:
   Entering Dialog; use Return[ ] to exit.
j$17 = xn$17 + i$17
```

`Stack[]` prints the tags that are associated with all rules whose evaluations are in progress.

```
Stack[]
{TraceDialog, Module, CompoundExpression, While,
  CompoundExpression}
```

An argument to `Stack`, if present, is a tag or pattern that constrains what is included in the stack dump.

This stack dump shows all `CompoundExpression` objects on the stack.

```
Stack[CompoundExpression]
{While[Abs[xn$17^2 - 2] > 10^-6,
    i$17 = 2/xn$17; j$17 = xn$17 + i$17; xn$17 = j$17/2];
  xn$17, i$17 = 2/xn$17; j$17 = xn$17 + i$17;
  xn$17 = j$17/2}
```

This stack dump shows only the `CompoundExpression` whose first part is a `Set`.

```
Stack[CompoundExpression[_Set, __]]
{i$17 = 2/xn$17; j$17 = xn$17 + i$17; xn$17 = j$17/2}
```

As you can see, you can get quite a bit of detail in this way. You can view *everything* on the stack using `Stack[_]`, which we will not do here in the interest of saving trees.

Abort the computation.

```
xn$17 := Abort[]
Return[]
TraceDialog::dgend: Exiting Dialog.
$Aborted
```

There are three functions that control what is placed on the evaluation stack.

- `StackBegin[`*`expr`*`]` starts a fresh stack to evaluate *`expr`*, so that calls to `Stack` within *`expr`* will not show any expressions that began evaluating before the call to `StackBegin`.
- `StackInhibit[`*`expr`*`]` evaluates *`expr`* without modifying the stack. This is a convenient way to keep the stack from becoming cluttered with evaluations that aren't germane to the computation being examined, such as those that are done within a dialog. (This explains why the default value for `TraceAction` is wrapped in `StackInhibit`.)
- `StackComplete[`*`expr`*`]` keeps earlier forms of expressions (which normally would be discarded) on the evaluation stack. During this call, the contents of the stack correspond to a trace with the options `TraceBackward->All` and `TraceAbove->True`.

`StackBegin/StackInhibit` can be used as "delimiters" for what appears on the stack. Here is an example that demonstrates quite concisely what `StackBegin` and `StackInhibit` do:

```
f[StackBegin[g[StackInhibit[h[Print[Stack[]]]]]]];
f[g]
```

13.2.4 TraceScan

`TraceScan[f, expr]` applies the function *f* to every intermediate expression in a trace before *expr* is evaluated. As usual, you can restrict the action of `TraceScan` by specifying a tag or pattern as an additional argument.

You can use `TraceScan` to cause some side effect such as keeping track of some statistics about a computation. For example, let's count the number of assignments to `xn` during the evaluation of `iterSqrt[2]`:

Note that the pure function completely ignores its argument.

```
c = 0;
TraceScan[c++ &, iterSqrt[2],
    xn = y_ /; NumberQ[Unevaluated[y]]
]
665857
------
470832
```

The count includes the assignment at the beginning of the `Module`.

```
c
5
```

Another possible application would be to abort a computation (by calling `Abort[]` from within *f*) if some criterion is true, such as exceeding a given amount of time, memory, iterations, etc.[3]

`TraceScan` applies *f* before the parts of the matched expression are evaluated. `TraceScan` takes an optional fourth argument, which is a function that is to be applied to the matched expression after the *parts* of the expression (but not the expression itself) have been evaluated.

Exercises

1. Use `TraceScan` to implement your own version of `TraceDialog`.
2. Implement a function that prompts the user every time a message is generated, asking whether or not to abort the computation. (Use the `Input` function to prompt the user.)

3. Note that there are built-in mechanisms for imposing each of these constraints that may be easier to use in many cases: the functions `MemoryConstrained` and `TimeConstrained`, and the global variables `$RecursionLimit` and `$IterationLimit`.

13.3 Additional Resources

For a novel application of `Trace`, see [Gayley 95], which describes an *execution profiler* for *Mathematica* programs. An execution profiler is a run-time tool that breaks down the execution time of a program into the parts that are attributable to particular statements or functions within the program. It is useful for locating performance bottlenecks.

[Smith 93] discusses the use of `TraceDialog` in some detail.

Christopher R. Stover has written a package called `traceUtil` that uses `TraceScan` to give the user detailed control over what parts of a trace are printed. It is available as *MathSource* item #0207-920.

Bibliography

[Aho & Ullman 77]

A.V. Aho and J.D. Ullman. *Principles of Compiler Design*. Addison-Wesley, Reading, MA, 1977.

[Blachman 92]

N. Blachman. *Mathematica: A Practical Approach*. Prentice-Hall, Englewood Cliffs, NJ, 1992.

[Clocksin & Mellish 84]

W.F. Clocksin and C.S. Mellish. *Programming in Prolog*, 2e. Springer-Verlag, Berlin, Germany, 1984.

[Cormen et al. 90]

T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. McGraw-Hill, New York, NY, 1990. (Also available from MIT Press, Cambridge, MA.)

[Fleck 76]

A.C. Fleck. On the impossibility of content exchange through the by-name parameter transmission mechanism. ACM *SIGPLAN Notices* 11(11):38–41.

[Gayley 93a]

T. Gayley. CleanSlate: A package for clearing symbols and freeing memory. *The Mathematica Journal* 3(3): 46–51. The *CleanSlate.m* package is available as *MathSource* item #0204-310.

[Gayley 94a]

T. Gayley (with J. Fultz, D. Withoff, and R. Villegas). Customizing the X front end, handling large lists, and generating surface plots. *The Mathematica Journal* 4(2):54–61.

[Gayley 94b]

T. Gayley. A *MathLink* program for accessing binary files. *The Mathematica Journal* 4(2):44–51.

[Gayley 94c]

T. Gayley. *A MathLink Tutorial*. Wolfram Research, Inc., Champaign, IL, 1994. Available as *MathSource* item #0206-693.

[Gayley 94d]

T. Gayley. *Tutorial: Package Design*. Wolfram Research, Inc., Champaign, IL, 1994. Available as *MathSource* item #0206-761.

[Gayley 95]

T. Gayley. A *Mathematica* profiler. *The Mathematica Journal* 5(3):48–60.

[Gaylord et al. 93]

R.J. Gaylord, S.N. Kamin, and P.R. Wellin. *Introduction to Programming with Mathematica*. TELOS (Springer-Verlag), Santa Clara, CA, 1993.

[Gray 94]

J.W. Gray. *Mastering Mathematica*. AP Professional, Cambridge, MA, 1994.

[Hayes 95]

A. Hayes. Experiments in efficient programming. *The Mathematica Journal* 5(1):24–31.

[Henderson 80]

P. Henderson. *Functional Programming: Application and Implementation*. Prentice-Hall, Englewood Cliffs, NJ, 1980.

[Jain 91]

R. Jain. *The Art of Computer Systems Performance Analysis*. Wiley, New York, NY, 1991.

[Jensen & Wirth 74]

K. Jensen and N. Wirth. *Pascal User Manual and Report*, 2e. Springer-Verlag, New York, NY, 1974.

[Keiper 93a]

J. Keiper. Numerical computation I. In *Selected Tutorial Notes*, Wolfram Research, Champaign, IL, 1993. Also available as *MathSource* item #0203-948.

[Keiper 93b]

J. Keiper. Numerical computation II. In *Selected Tutorial Notes*, Wolfram Research, Champaign, IL, 1993. Also available as *MathSource* item #0203-937.

[Kernighan & Ritchie 78]

B.W. Kernighan and D.M. Ritchie. *The C Programming Language*. Prentice-Hall, Englewood Cliffs, NJ, 1978.

[MacLennan 83]

B.J. MacLennan. *Principles of Programming Languages: Design, Evaluation, and Implementation*. Holt, Rinehart & Winston, New York, NY, 1983.

[Maeder 91]

R. Maeder. *Programming in Mathematica*, 2e. Addison-Wesley, Redwood City, CA, 1991.

[Maeder 94a]

R. Maeder. *The Mathematica Programmer.* AP Professional, Cambridge, MA, 1994.

[Maeder 94b]

R. Maeder. Logic programming I: The interpreter. *The Mathematica Journal* 4(1):53–63.

[Naur 63]

P. Naur, ed. Revised report on the algorithmic language Algol-60. *Comm. ACM* 6(1):1–17.

[Novak 94]

J. Novak. *Tutorial: Notebooks for Integrated Applications.* Wolfram Research, Inc., Champaign, IL, 1994. Available as *MathSource* item #0206-671.

[Shaw & Tigg 94]

W.T. Shaw and J. Tigg. *Applied Mathematica: Getting Started, Getting It Done.* Addison-Wesley, Reading, MA, 1994.

[Smith 93]

B. Smith. Tracing and debugging *Mathematica* programs. In *Selected Tutorial Notes*, Wolfram Research, Inc., Champaign, IL, 1993. Also available as *MathSource* item #0203-858.

[Smith & Blachman 94c]

C. Smith and N. Blachman. *The Mathematica Graphics Guidebook.* Addison-Wesley, Reading, MA, 1994.

[Steele 84]

G.L. Steele, Jr. *Common Lisp.* Digital Press, Burlington, MA, 1984.

[Stroustrup 91]

B. Stroustrup. *The C++ Programming Language*, 2e. Addison-Wesley, Reading, MA, 1991.

[Wagner 95]

D.B. Wagner. Power programming: Dynamic programming. *The Mathematica Journal* 5(4):42–51.

[Wagner 96a]

D.B. Wagner. Power programming: Dependency analysis. *The Mathematica Journal* 6(1):54–65.

[Wagner 96b]

D.B. Wagner. Power programming: *MathLink* mode. *The Mathematica Journal* 6(3):38–51.

[Wagon 91]

S. Wagon. *Mathematica in Action.* W.H. Freeman, New York, NY, 1991.

[Wickham-Jones 94]

T. Wickham-Jones. *Mathematica Graphics: Techniques and Applications*. TELOS (Springer-Verlag), Santa Clara, CA, 1994.

[Withoff 93]

D. Withoff. *Mathematica* internals. In *Selected Tutorial Notes*, Wolfram Research, Inc., Champaign, IL, 1993. Also available as *MathSource* item #0203-982.

[Wolfram 91]

S. Wolfram. *Mathematica: A System for Doing Mathematics by Computer*, 2e. Addison-Wesley, Reading, MA, 1991.

[Wolfram 96]

S. Wolfram. *The Mathematica Book*. Wolfram Media, Inc., Champaign, IL, 1996.

[WRI 91]

Mathematica Warning Messages. Technical Report, Wolfram Research, Inc., Champaign, IL, 1991. Also available as *MathSource* item #0203-612.

[WRI 93a]

Major New Features in Mathematica Version 2.2. Technical Report, Wolfram Research, Inc., Champaign, IL, 1993. Also available as *MathSource* item #0204-804.

[WRI 93b]

Guide to Standard Mathematica Packages Version 2.2. Technical Report, Wolfram Research, Inc., Champaign, IL, 1993. Version 2.1 of this report is available as *MathSource* item #0204-365.

[WRI 93c]

MathLink Reference Guide Version 2.2. Technical Report, Wolfram Research, Inc., Champaign, IL, 1993.

[WRI 93d]

Guidelines for Mathematica Documentation, Packages, and Notebooks. Technical Report, Wolfram Research, Inc., Champaign, IL, 1993. Available as part of *MathSource* item #0204-961.

Index

Symbols

!! (show file) 388
! (Factorial) 31
(SlotSequence) 197
(Slot) 102
$Aborted 378
$Context 230
$ContextPath 234
$DefaultFont 179
$Display 396
$DisplayFunction 396
$Failed 404
  returned from installable functions 345
$HistoryLength 58
$IterationLimit 29
$MachinePrecision 20
$Messages 258
$Output 395
$Packages 239
$Path 239
$Post 281
$Pre 281, 292
$PrePrint 281
$PreRead 281
$RecursionLimit 152
$SystemID 345, 392
$TracePattern 416
% (previous result) 33
& (Function) 102
() 31
* (string wildcard) 46, 64
* (Times) 30
+ (Plus) 30
// (postfix function application) 37
//. (ReplaceRepeated) 152
/: ... := (TagSetDelayed) 177
/: ... = (TagSet) 177
/@ (Map) 62, 98
:= (SetDelayed) 33
:> (RuleDelayed) 147
<< (Get) 388
<> (StringJoin) 65
=!= (UnsameQ) 32
= (Set) 33
=. (Unset) 151
=== (SameQ) 32
>> (Put) 388
>>> (PutAppend) 388
? (help) 46
@ (prefix function application) 37
@@ (Apply) 98
[[ ]] (Part) 42
^ (Power) 30
^:= (UpSetDelayed) 177
^= (UpSet) 177
` (context mark) 230
{} (list braces) 49
| (Alternatives) 164
~ (infix function application) 37
- (Subtract) 30
-> (Rule) 42

A

addition 30
And 32
Annotation 243
AppendTo 59
Apply 98
approximate numbers 20
arithmetic
  arbitrary-precision 20
  exact 20
  operators 30

Array 52
assignment operators 33
atoms 17, 19
Attributes 27, 56, 215
attributes
    Constant 190
    Flat 190
    HoldAll 27, 90, 190
    HoldFirst 90, 190
    HoldRest 90, 190
    Listable 56, 190
    Locked 179, 190
    NumericFunction 190, 269
    OneIdentity 190
    Orderless 190
    Protected 175, 179, 191
    of pure functions 196
    ReadProtected 191, 239
    Stub 191
    table of 190
    Temporary 191

B

backtracking 150
base conversion 117
Begin 231
BeginPackage 237
binary trees 122
binding (a.k.a. scoping) 93
bisection 87, 369
Blachman, N.R. 246
Blank 39, 72
BlankNullSequence 158
BlankSequence 158
Block 94
    as an alternative to Hold 224
block-structured languages 96
box representation 277
Break 87
Bruck, R. xvi
bubblesort 159
bug report
    how to submit xv

C

call by name 89
call by reference 92
call by value 89
CallPacket 344
Catalan 23
Catch 117
CForm 277
Characters 66
character strings 24, 63
    and *MathLink* 381
Clear 34, 72
ClearAll 56, 90, 262
Close 393
CommaDelimitedForm 391
Compile 325
CompiledFunction 325
compiled functions 325
    parameter specification 327
Complement 61
complex numbers 19
    in compiled functions 328
CompoundExpression 40, 76
computational complexity 306
conditions 154
cons (Lisp operation) 308
Constant 190
Context 230
context 229
    changing 231
    Global` 230
    mark 230
    private 237
    search path 234
    System` 230
Contexts 232
ContextToFilename 239
Continue 87
control flow 83
Cormen, T.H. 165
Count 57

D

debugging 405
    interactive 413

MathLink programs 349, 356
DeclarePackage 247
DefineExternal 353
Degree 23
DelimitedForm 391
derivative function 156
destructuring 143
DiagonalMatrix 54
Dialog 413
Dimensions 56
Directory 387
DirectoryStack 387
Dispatch 153
Display 395
DisplayFunction 395
Divide 31
divide and conquer 127
Do 84
dot product 100
DownValues 148, 177, 186
Drop 60
dynamic programming 165

E

E 23
End 231
Equal 32
error messages 237
Euclidean distance 100
EulerGamma 23
Evaluate 201
EvaluatePacket 371
evaluation
  of expressions 25
  nonstandard 27, 197
  process 185
  stack 418
  standard 26
exact arithmetic 20
ExactNumberQ 267
exact numbers 20
Expand 175
ExpandAll 176
exponentiation 30
expressions
  atomic 17, 19
  compound 39
  evaluation of 25, 185
  head of 132
  normal 17
  as trees 26
ExternalCall 344
Extract 210

F

Factorial 31
Fibonacci numbers 166
files
  .mx 391
  .tm 343
files and directories 387
file search path 239
finite state machine 123
FixedPoint 114
FixedPointList 115
Flat 190
Flatten 60
Fold 116
FoldList 116
For 85
Format 274
FormatType 394
FormatValues 186
FortranForm 277, 390
FreeQ 156
FromCharacterCode 66
FullForm 18, 21, 131
Function 101–102
functional programming 97
Function application
  infix 37
  postfix 37
  prefix 37
functions 71
  compiled 325
  defining 35, 38, 71
  local variables 76
  overloading 75
  parameters to 35, 72
  pure 101

recursive 127
type checking 74
virtual 178

G

Gayley, T. 365, 378, 384
Gaylord, R.J. 184
Get 239, 388
GoldenRatio 23
Goto 88
Greater 32

H

Harris, J. 282
hashing 311
Hayes, A. 224
Head 132
HeldPart 210
help 46
Hold 27
HoldAll 27, 90, 190, 198
HoldAllComplete 200
HoldFirst 90, 190, 198
HoldForm 28
HoldPattern 148, 204
HoldRest 90, 190, 198
Horner's rule 116
Huffman coding 118

I

IdentityMatrix 54
If 81
imperative programming 141
indexed variables 311
InexactNumberQ 267
Input 209
input/output 387
InputForm 21, 277, 393
InputStream 402
Install 343
IntegerDigits 337
IntegerList 346
integers 19
Intersection 61
iteration limit 29

J

Jain, R. 237
Jensen's device 91
Jentschura, U. 248
Join 61

K

Kamin, S.N. 184

L

lambda expressions (Lisp) 102
Larson, J. xvi
Leiserson, C.E. 165
Length 56
LessEqual 32
level specifications 104
LinkClose 337
LinkConnect 350
LinkConnectedQ 352
linked lists 308
LinkLaunch 337
LinkObject 345
LinkPatterns 344
LinkRead 337–338
LinkReadHeld 352
LinkReadyQ 352
Links 345
LinkWrite 338
LinkWriteHeld 352
Lisp 308
and *Mathematica* compared 139
Listable 54, 190
lists 41, 49
applying functions to 61
extracting elements from 60
generating 52
indexing 42
linked 308
modifying in place 63
nested 49–50
set operations on 61
structural operations on 58
subscripting 50
unnesting 60
Literal 148, 204

local variables 76
Locked 179, 190
loops 83

M

MachineNumberQ 21
machine precision numbers 20
Maeder, R. 149, 184, 246
magic cookies 207
Map 62, 98
MapAll 105
MapAt 176
MapThread 107
MathLink 335
  arrays 380
  character encodings 381
  disowning memory 364
  low-level errors 374
  packets 371
  template files 341
  textual interface 379
matrices 50
  diagonal 54
  identity 54
  Toeplitz 112
  transposing 51
MatrixForm 50
MatrixQ 58
mcc 341
mean
  arithmetic 98
  geometric 100
  harmonic 100
MemberQ 57
memoization (result-caching) 165
mergesort 130, 305
  tracing 410
MLCheckFunction 364
MLClearError 375
MLClose 336
MLDeinitialize 336
MLDisownIntegerList 363
MLError 375
MLErrorMessage 375
MLEvaluate 369, 371
MLEvaluateString 375
MLGetByteString 382
MLGetByteSymbol 383
MLGetFunction 364
MLGetInteger 337
MLGetNext 336–337
MLGetType 376
MLGetUnicodeSymbol 383
MLInitialize 336
MLMain 342
MLNewPacket 374
MLNextPacket 371
MLOpenArgv 336
MLPutByteString 382–383
MLPutByteSymbol 383
MLPutCharToString 383
MLPutFunction 360
MLPutInteger 337
MLPutString 336
MLPutUnicodeSymbol 383
MLStringChar 383
Module 76
mprep 341
multiplication 30

N

N 22, 266
Names 179, 214
Needs 239, 388
Nest 114
NestList 114
Newton's method 116
NonNegative 154
nonstandard evaluation 27, 197
normal expressions 17
NullWords 402
numbers 19
  approximate 20
  exact 20
  machine precision 20
numerical evaluation 266
  preventing 269
numeric constants 23
numeric expressions 24
NumericFunction 190, 269

NumericQ 24, 267
NValues 186, 271

O

OneIdentity 190
OpenAppend 392
OpenWrite 392
operators
  arithmetic 30
  assignment 33
  relational and logical 31
Options 43, 162
options
  to functions 43
  implementing 161
Or 32
Orderless 190
OutputForm 277, 393
output formats
  box representation 277
  CForm 277
  CommaDelimitedForm 391
  customizing 274
  DelimitedForm 391
  FortranForm 277, 390
  InputForm 277, 393
  MatrixForm 50
  OutputForm 277, 389, 393
  Overscript 274
  SequenceForm 275
  SpaceDelimitedForm 390
  StandardForm 277, 393
  StringForm 275
  Subscript 274
  Subscripted 274
  Superscript 274
  TableForm 50, 389
  TeXForm 394
  TraditionalForm 277, 393
OwnValues 186

P

packages 229
  annotations and keywords 243
  AntiShadow 249
  CleanSlate 244
  DependencyAnalysis 223
  importing other packages 240
  LCGRandom 237
  Unshadow 248
  Utilities`Package` 243
parameters 72
parent-child connection 349
parentheses 31
parser traps 44
patterns 141
  blank sequences 158
  constraining 153
  default values for 155
  naming 162
  repeated 163
peer-to-peer connection 349
Pi 23
Position 57, 134, 137
Precision 20
predicates 57
PrependTo 59
Prolog 150
Protected 175, 179, 191
pure functions 101
  and held expressions 213
Put 388
PutAppend 388

Q

quicksort 310

R

Random 36
Range 52
rank (of a tensor) 332
rational numbers 19
Read 403
reading
  comma-delimited data 400
  numbers 398
  strings 399
ReadList 396
ReadProtected 191, 239
RealList 346

real numbers 19
    in compiled functions 328
RecordSeparators 399
recursion 127
ReleaseHold 67
Remove 235
ReplaceRepeated 152
ResetDirectory 388
result caching 165
Return 87
Reverse 60
Rivest, R.L. 165
RotateLeft, RotateRight 60
Rule 42
rule-based programming 141
RuleDelayed 147, 200
rules, types of 186
run-length encoding 163

S

SameQ 32
Save 391
Scan 112
scoping (of symbol names) 93
sea shore
    selling sea shells by 118
Select 57
Sequence 193, 196
SequenceHold 200
sequence splicing 196
Set 33, 78, 200
SetAttributes 56
SetDelayed 33, 78, 200
SetDirectory 387
SetOptions 43, 162
sets (mathematical) 61
SetStreamPosition 403
shadowing 11, 246
side effects 28
Sort 60
SpaceDelimitedForm 390
special input forms 18, 30
sprintf (C library function) 361
Sqrt 22
Stack 418
StackBegin 419
StackComplete 419
StackInhibit 419
standard evaluation 26
StandardForm 277, 393
stdlink 358
Stover, C.R. 421
stream 392
StreamPosition 403
Streams 392
StringJoin 65
StringLength 63
string patterns 214
StringPosition 63
StringReplace 65
strings (character) 24, 63
Stub 191
subscripting lists 50
Subtract 31
subtraction 30
SubValues 186
Switch 82
symbolic head 193
symbols 19
    information about 46

T

Table 53
TableForm 50, 389
TableSpacing 389
TagSet 177
TagSetDelayed 177
Take 60
template file 341
Temporary 191
tensors 332
TeXForm 277, 394
Thickness 179
Thread 214
Through 112, 151
Throw 117
ToCharacterCode 66
ToExpression 66, 209
ToHeldExpression 67, 209
tournament algorithm 131

Trace 25
TraceAction 418
TraceBackward 410
TraceDepth 411
TraceDialog 413
TraceForward 409
TraceOff 410
TraceOn 410
TraceOriginal 412
TracePrint 413
TraditionalForm 277, 393
Transpose 51
TreeForm 26
trees (graph theory) 118
type checking 74

U

Unequal 32
Unevaluated 203
Uninstall 345
Union 61
UnsameQ 32
Unset 151
Update 192
UpSet 177
UpSetDelayed 177
UpValues 177, 186
upvalues
    versus downvalues 178
    limitations of 178
    compared to virtual functions 178
usage messages 237

V

vector arithmetic 55
VectorQ 58
Verbatim 206
Villegas, R. 284
virtual function 178

W

Wagon, S. 138
Wellin, P.R. 184
Which 83
While 86
wildcard characters 214
wildcard searches 46, 64
WordSeparators 400
Write 393
WriteString 393

X

Xor 38

Z

Zizza, F. 304

About the Author

David B. Wagner is the president of Principia Consulting, which provides training and consulting on the *Mathematica* system for doing mathematics by computer. He also is a columnist for *The Mathematica Journal* and a consulting instructor at the *Mathematica* Training Center of Wolfram Research, Inc. Previously, he was a faculty member in the Department of Computer Science at the University of Colorado, where he used *Mathematica* in his research on computer systems performance analysis. Dr. Wagner holds a Ph.D. in Computer Science from the University of Washington.

SOFTWARE AND INFORMATION LICENSE

The software and information on this diskette (collectively referred to as the "Product") are the property of The McGraw-Hill Companies, Inc. ("McGraw-Hill") and are protected by both United States copyright law and international copyright treaty provision. You must treat this Product just like a book, except that you may copy it into a computer to be used and you may make archival copies of the Products for the sole purpose of backing up our software and protecting your investment from loss.

By saying "just like a book," McGraw-Hill means, for example, that the Product may be used by any number of people and may be freely moved from one computer location to another, so long as there is no possibility of the Product (or any part of the Product) being used at one location or on one computer while it is being used at another. Just as a book cannot be read by two different people in two different places at the same time, neither can the Product be used by two different people in two different places at the same time (unless, of course, McGraw-Hill's rights are being violated).

McGraw-Hill reserves the right to alter or modify the contents of the Product at any time.

This agreement is effective until terminated. The Agreement will terminate automatically without notice if you fail to comply with any provisions of this Agreement. In the event of termination by reason of your breach, you will destroy or erase all copies of the Product installed on any computer system or made for backup purposes and shall expunge the Product from your data storage facilities.

LIMITED WARRANTY

McGraw-Hill warrants the physical diskette(s) enclosed herein to be free of defects in materials and workmanship for a period of sixty days from the purchase date. If McGraw-Hill receives written notification within the warranty period of defects in materials or workmanship, and such notification is determined by McGraw-Hill to be correct, McGraw-Hill will replace the defective diskette(s). Send request to:

Customer Service
McGraw-Hill
Gahanna Industrial Park
860 Taylor Station Road
Blacklick, OH 43004-9615

The entire and exclusive liability and remedy for breach of this Limited Warranty shall be limited to replacement of defective diskette(s) and shall not include or extend to any claim for or right to cover any other damages, including but not limited to, loss of profit, data, or use of the software, or special, incidental, or consequential damages or other similar claims, even if McGraw-Hill has been specifically advised as to the possibility of such damages. In no event will McGraw-Hill's liability for any damages to you or any other person ever exceed the lower of suggested list price or actual price paid for the license to use the Product, regardless of any form of the claim.

THE McGRAW-HILL COMPANIES, INC. SPECIFICALLY DISCLAIMS ALL OTHER WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO, ANY IMPLIED WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Specifically, McGraw-Hill makes no representation or warranty that the Product is fit for any particular purpose and any implied warranty of merchantability is limited to the sixty day duration of the Limited Warranty covering the physical diskette(s) only (and not the software or in-formation) and is otherwise expressly and specifically disclaimed.

This Limited Warranty gives you specific legal rights; you may have others which may vary from state to state. Some states do not allow the exclusion of incidental or consequential damages, or the limitation on how long an implied warranty lasts, so some of the above may not apply to you.

This Agreement constitutes the entire agreement between the parties relating to use of the Product. The terms of any purchase order shall have no effect on the terms of this Agreement. Failure of McGraw-Hill to insist at any time on strict compliance with this Agreement shall not constitute a waiver of any rights under this Agreement. This Agreement shall be construed and governed in accordance with the laws of New York. If any provision of this Agreement is held to be contrary to law, that provision will be enforced to the maximum extent permissible and the remaining provisions will remain in force and effect.