EXTENSION OF MATHEMATICA SYSTEM FUNCTIONALITY

VICTOR ALADJEV

Victor Aladjev, Vjacheslav Vaganov

Extension of Mathematica system functionality

Tallinn – 2015

Extension of Mathematica system functionality: Victor Aladjev, Vjacheslav Vaganov.– Tallinn: TRG Press, 563 p., 2015

Systems of computer mathematics find more and more broad application in a number of natural, economical and social fields. These systems are rather important tools for scientists, teachers, researchers and engineers, very well combining symbolical methods with advanced computing methods. One of leaders among means of this class undoubtedly is the***Mathematica*** system. The book focuses on one important aspect–*modular programming* supported by***Mathematica.*** The given aspect is of particular importance not only for appendices but also above all it is quite important in the creation of the user means that expand the most frequently used standard means of the system and/or eliminate its shortcomings, or complement the new facilities.

Software tools presented in the book contain a number of rather useful and effective methods of*procedural* and*functional* programming in***Mathematica*** system that extend the system software and allow sometimes much more efficiently and easily to program the objects for various purposes first of all wearing system character. The above software tools rather essentially dilate the***Mathematica*** functionality and can be useful enough for programming of many applications above all of system character. Furthermore, the book is provided with freeware package***AVZ_Package*** containing more than***680*** procedures, functions, global variables and other program objects. The present book is oriented on a wide enough range of users of systems of the computer mathematics, teachers and students of universities at courses of computer science, mathematics and other natural–science disciplines.

In the course of preparation of the present book the license releases***8÷ 10*** of the***Mathematica*** system provided by*Wolfram Research Inc*. have been used.

Contents

Mathematica 2, 5 ÷ 10 – trademarks of Wolfram Research Inc.

Preface

*Systems of computer mathematics (**SCM**)* find more and more wide application in a number of natural, economical and social sciences such as**:***informatics, chemistry, mathematics, physics, technologies, education, economics, sociology, etc*. Such systems as***Mathematica, Maple, REDUCE, MuPAD, Derive, Magma, Axiom, Maxima, GAP, MathPiper*** and others are more and more demanded for learning of the mathematically oriented disciplines, in various scientific researches and technologies. These systems are the main tools for teachers, scientists, researchers, and engineers. Researches on the basis of technology ***SCM,*** as a rule, well combine algebraic methods with advanced computing methods. In this sense of***SCM –*** interdisciplinary area between informatics and mathematics in which researches are concentrated on development of algorithms for algebraical*(symbolical)* and numerical calculations and data processing, and on creation of programming languages along with program environment for realization of this kind of algorithms and tasks of different purpose which are based on them.
Solution of applied user problems in one or the other field of appendices is supported by*packages of applied programs (**PAP**or simply packages)* of special, highly specialized or general purpose. Classification and characteristic of such class of software can be found in our previous books [3–5]. Naturally, the qualified user well owning one of effective programming languages*(for example, Basic, C, Fortran, PL/1, Pascal, Lisp, Prolog, etc.)* in a number of cases for the solution of own tasks can independently write and debug a separate program or a complex of programs allowing to realize algorithm of its tasks on a personal computer. In some cases such approach can be more effective, than use for these purposes of ready software since the software developer at the same time well owns specifics of the solved task and conditions of its operation. However, such approach demands as a rule of serious costs and at present abundance of various type and purpose of means for a personal computer becomes considerably inexpedient. At the same time, developed ***PAP*** are supplied with own builtin programming language of one or other level of complexity allowing to program the whole tasks or their separate fragments which may be in the environment of a package are inefficiently, inexpedient, and in some cases and is impossible to realize by the standard means of a package.

This book is devoted to the class of software called by ***systems of computer mathematics*** which, first of all, are intended for the solution of problems of mathematical character, and, first of all, to leaders in this class to systems ***Mathematica*** and***Maple.*** Moreover, only the indirect attention concerning comparison of systems on certain separate moments is paid to the second system whereas quite developed their comparative analysis can be found in our books [28-30]. At that, much attention was paid both on experience with described means, and features of their usage, and also recommendations for the user following from them. As far as possible, the most effective*technique* of application of these means for the solution of those or other applied user tasks have been offered. Moreover, in book [33] we presented an excursus in history of computer algebra systems that represents a certain interest for the user of this class of software. Rather detailed characteristic of this series of books can be found, in particular, in [30-33] and in the present book isn't considered. Our operating experience with*systems of computer algebra*, first of all,***Mathematica*** and***Maple*** allowed not only to carry out a comparative analysis of these means, to reveal deficiencies inherent to them, and also to create a number of the means expanding their functionality and eliminating their some defects. All these questions including questions of*terminological* character with various extent of detailing have been considered in a series of our books and papers [1-48].

The ***Mathematica*** system along with the above–mentioned***Maple*** system is one of the most known and popular***SCM,*** it contains a rather large number of functions for providing as symbolical transformations, and for numerical calculations. The***Mathematica*** system for today is multipurpose means that includes a large number of opportunities for the solution of quite wide range of problems. Naturally, for these means can`t be given a rather full analysis within the framework of the given book. Furthermore, the target of the book consists in other– in the book the attention is focused only on one aspect of system– opportunities of her program environment for solution of special problems of mass and system character.

This aspect has the special importance not only for solution of applied tasks but above all it is quite important at creation of the software expanding often used system means and/or eliminating their defects, or supplementing the system with new means. In this context possibilities of built–in language of the system on creation of such kind of procedures or functions are of special interest. So, programming in the system is a multifaceted subject and in it we focus attention only on questions of realization of procedures/functions that represent main program objects both for the most often used means of the user, and for the means expanding and improving standard system means in the system software, i.e. realized by means of the built–in language of the system*(**Math**language)*. In this context it is also possible to estimate in quite full measure the***Mathematica*** system software, without regarding to some subjective moments, first of all, the user preferences and habits. Naturally, these moments play a rather essential part for the user which has a certain experience of work with program languages of procedural type whereas for a beginner they stand not so sharply because of lack of such experience. So, considering orientation of the given book, for conscious acquaintance with its contents the knowledge of***Math***language at the level above the initial is supposed, for example, within the works [29-33,51,52,55,57,60,62,64,66,71]. Since the***10th*** version***Math***–language is called as Wolfram Language what, in our opinion, is result of certain painful ambitions similar to those that are associated with book***”****A New Kind of Science****”*** along with a fair share of self– advertisement of allegedly new means.
The given book affects a rather extensive material on***Mathematica*** software in the context of its opportunities in*procedural* and*functional* programming. Meanwhile, main purpose of this book laid aside the questions which are of interest, first of all, to readers who are in own activity at the first stages of an mastering of the***Mathematica*** system. For beginners it is recommended to address oneself to the corresponding editions whose list is rather extensive, above all, the English-language. The***Mathematica*** system is considered and in Russian–language literature, however English–language editions, in our opinion, are represented to us more preferable. In general, it is possible to familiarize oneself with literature on the website***www.wolfram.com/books,*** quite useful sources can be found in the represented references, including a rather useful references in the*Internet****.***
Thus, the given book represents a certain set of the selected system problems whose

purpose not only to expand or make more effective the***Mathematica*** system, but also to give certain help to those users of the***Mathematica*** who would like to move from the user's level to a level of the programmer or to those who when using***Mathematica*** already faced some its restrictions and want to improve its program environment. At that, the skilled***Mathematica*** programmers probably will also be able to find for themselves in our book a rather useful information and of applied character, and to reflection. Therefore illumination only of some questions essence without their rather detailed discussion, certain nuances and consideration of adjacent questions that are often interesting and important per se often takes place. Moreover, the system means presented in the book can be used as rather useful means at developing own applications in the environment of***Mathematica.*** In our opinion, an analysis of the source codes of the means presented in this book which use both effective, and nonstandard methods of programming along with quite certain practical interest will allow to master the environment of ***Mathematica*** system more deeply. For convenience of their use in the given quality the reader has possibility of free download of***AVZ_Package*** package for***Mathematica*** system of versions***8÷10*** which contains these means [48]. The means considered throughout the present book answer fully the main goal of the offered book which can be characterized by the following***2*** main directions, namely**:**

(1) representation of a number of useful enough means of system character that expand and supplement standard means of the***Mathematica*** system**;** ***(2)*** illustration on their example of receptions and methods, enough useful in*procedural* and*functional* programming, along with a number of essential enough features of this paradigm of programming in the conditions of the program environment of the***Mathematica*** system. Here is quite appropriate to note a quite natural mechanism of formation of own software means of the user working in some program environment. In course of programming of one or other means, or the whole project a certain situation is quite real when is rather expedient to program some additional tools that are absent among standard means, either they are more effective, or they are more convenient than standard means. In many important cases the applicability of these means can have mass enough character, allowing to form program toolkit of quite wide range of applicability.

Exactly in many respects thanks to the*described* mechanism we have created quite famous library***UserLib*** for***Maple*** along with package***AVZ_Package*** for***Mathematica*** which contain more than***850*** and***680*** means respectively [47,48]. All above-mentioned means are supplied with*FreeWare* license and have open program code. Such approach to programming of many projects both in***Mathematica,*** and in***Maple*** also substantially promoted emergence of a number of system means from above–mentioned library and package, when development of software for simplification of its realization revealed expediency of definition of the new accompanying tools of system character that are rather frequently used both in applied and in system programming. So, openness of the***AVZ_Package*** package code allows both to modify the means containing in it, and to program on their basis own means, or to use their components in various appendices. In our opinion, tasks and means of their realization in***Mathematica*** which are presented in the above package can be rather useful at deeper mastering of system and in a number of cases will allow to simplify rather significantly programming of appendices in it, first of all, the system problems. At that, the methodological considerations represented in our

previous books [29-33] fully remain in force and relative to the present book.
Means of***AVZ_Package*** package have different complexity of organization and used algorithms**;** in certain cases, they use effective and nonstandard receptions of programming in***Mathematica.*** The given means can be used as individually*(for the decision of various problems or for creation on their basis of new means)****,*** and in structure of***AVZ_Package*** package extending standard tools of the***Mathematica,*** eliminating a number of its defects and mistakes, raising its compatibility relatively to its releases and raising effectiveness of programming of problems in***Mathematica.*** A tool represented in the book is supplied with description and explanations, contains the source code and the more typical examples of its application. As required, a description has supplied by necessary considerations, concerning peculiarities of program execution in the***Mathematica*** environment.
The given book considers certain principal questions of*procedure–functional* programming in***Mathematica,*** not only for the decision of various applied problems, but, first of all, for creation of the software expanding frequently used facilities of the system and/or eliminating their defects or expanding the system with new facilities. The software presented in this book contains a series of useful and effective receptions of programming in***Mathematica*** system**,** and extends its software which enables more simply and effectively to programme in the system***Mathematica*** the problems of various purpose. The represented monograph, is mostly for people who want the more deep understanding in the ***Mathematica*** programming, and particularly those ***Mathematica*** users who would like to make a transition from the user to a programmer, or perhaps those who already have certain limited experience in***Mathematica*** programming but want to improve their possibilities in the system. Whereas the expert***Mathematica*** programmers will also probably find an useful enough information for yourself.

At that, it should be noted that the source codes of means given in this book contain calls of non–standard tools that didn**'**t find reflection in the present book in a number of cases, but are presented in our package[48]. Therefore, their detailed analysis requires acquaintance with these tools, at least, at the level of usages on them. Meanwhile, the main algorithm of many means of the presented book is rather well looked through and without acquaintance with similar means while real use of these means perhaps only after loading of this package into the current session. Along with the illustrative purposes the means represented in this monograph quite can be used and as enough useful means extending the program**Mathematica** environment that rather significantly facilitate programming of a wide range of the problems first of all having the system character. Our experience of conducting of the master classes of various level in systems and**Mathematica,** and**Maple** confirms expediency of application in common with standard means of both systems and some user tools created in the course of programming of appendices. Tools represented in the book increase the range and efficiency of usage of ***Mathematica*** on Windows platform owing to the innovations in three basic directions, namely**:*(1)****elimination of a series of basic defects and shortcomings,* ***(2)****extending of capabilities of a series of standard tools*, and***(3)****replenishment of the system by new means which increase capabilities of its program environment, including the means which improve the level of compatibility of releases****7 – 10.*** At last, with organization of the user software and programming of large-scale systems in***Mathematica*** software along with our standpoint on a question**:** ***Mathematica*** or***Maple?*** the interested reader can familiarize in

[29–33]. At last, a number of means represented in the above books is intended for a extension of standard means of the systems***Mathematica*** and***Maple*** along with elimination of their shortcomings and mistakes. These means not only more accurately accent distinctions of both systems, but also their problems of common character. And in this relation they allow to look from different points of view on these or other both advantages, and shortcomings of both systems. In the present book we present a number of means of similar type concerning the**Mathematica** system. At that, it should be noted that a mass optimization of procedures have not been performed, procedures in many cases have been written, as they say on**'***sheet***';** on the other hand, numerous procedures have been optimized using both the standard means and newly created tools of system character. In this context here there is a magnificent experimental field for increasing of professionalism of the user at operating with the**Mathematica** software.

Inclusion of source codes of the procedures and functions presented in this book with their short characteristic directly in the book text allows to work with them without computer, considering a habit of considerable number of the users of the senior generation to operate with program listings before exit to the computer what in a series of cases promoted better programming in due time at programming in batch mode. In our opinion, skill to operate with program listings is a rather important component of the programmer culture, allowing better to feel the used program environment. In a certain measure it is similar to possession of the musician by the sheet music.

Moreover, many listings of the represented means have a rather small size, allowing to analyze them outside of the**Mathematica** environment in the assumption that the reader is sufficiently familiar with its software. Now, at mass existence of personal computers of various type the mentioned visual analysis of the program listings was replaced with the mode of interactive programming, however it**'**s not the same, and in the first case the process of programming seems to us more better and efficient. Meanwhile, even tools with small source code often are useful enough at programming of various applications, in particular, of system character. Whereas others demand for the understanding of serious enough elaboration, including acquaintance with our package***AVZ_Package*** [48].

As shows our experience, the programming in the above mode slightly more slowly, than directly on the computer, however it allows to concentrate our better on an object of programming and it is better to think over a problem and a way of its decision, rather, than method of its decision in the so-called interactive mode. Even in the presence of the*personal computer (***PC***)* we got used the*basic skeleton* of a program to write on paper and only then to pass to debugging onto the personal computer in the interactive mode. So, in our opinion, such approach allows to write programs more thoughtfully**;** at that, following the old habit to write optimal enough codes for their subsequent performance on quite limited computing resources of the computers***20–30*** years ago. However, in many respects this is matter of habit, however you shouldn**'**t forget that the old isn**'**t always worse than new one and, getting new opportunities, we, often, lose the old skills important for work. Here and in this case, having received very convenient means of communication, we, sometimes, lose sight of efficiency of a program code, creating it without especial difficulties in the interactive mode with the only purpose to receive the demanded result, often, ignoring quality.

Of course, there is no only best way of creation of epy programs. Different technologies and paradigms are required for the programming of different problems and their levels of complexity. So, in the elementary cases is quite enough of the knowing of elements of structural writing of programs. While for creation of complex program projects is required not only to be fluent in a programming language in rather full volume, but also to have notion of the principles of elaboration and debugging of programs, opportunities of both standard and other libraries of one or the other software, etc.

As a rule, than the problem is more complex, the more time is required for mastering of the tools necessary for its decision. In this context the software *(procedures/functions/global variables)* which is presented in the present book contain a number of rather useful and effective methods of programming in the**Mathematica** environment and extends its program environment, they give opportunity more simply and effective to program different problems. These means in the process of application of the***AVZ_Package*** package are updated, taking into account both the new means, and the optimization of already existing means. In many problems of different purpose the package ***AVZ_Package*** showed itself as a rather effective toolkit. The package on the freeware conditions is attached to the present book [48].

Chapter 1. Additional means in interactive mode of the *Mathematica*system

Further we will distinguish two main operating modes with ***Mathematica* –** *interactive* and*program*. Under the first mode step-by-step performance with a***Mathematica*** document, i.e. from an input**In[*n*]** up to output**Out[*n*]** will be understood while under the*program* mode the operating within a*block* or a*module* is understood. In the present chapter some additional means rather useful at work with***Mathematica*** in interactive mode are considered. In the course of operating in*interactive* mode in many cases there is a need of use of earlier calculated expressions in the previous**In**-paragraphs. For this purpose the**%*k***operator {**%, %%, %% … %%*(k****times)*} serves which defines return of the last, penultimate and***k***th previous result of calculations in the current session. In addition, it should be noted that**%–**operators in systems ***Mathematica*** and***Maple*** are conceptually various. Though, having various real areas of applicability in***Mathematica*** and***Maple,*** at the same time**%–** operators possess both the shortcomings, and essential advantages [28-33]. The***Mathematica*** supports***2*** rather useful predetermined global variables**:**
$Line–*defines number of the last***In***paragraph of the current session**;***
$HistoryLength–*defines number of the previous paragraphs***In***and***Out***kept in the current session*.
Moreover, these variables allow redefinitions by simple assignment of new values. For**$HistoryLength** variable value by default is the*infinity***(***∞***);*** but using smaller installations for the variable, it is possible significantly to save the size of*RAM* required for***Mathematica*** system. In turn, global variable **$Line1** unlike the standard global variable**$Line** determines total number of**Out**paragraphs of the current session, including results of calculation of the user packages loaded into the session from files of formats {**”**cdf**“, “**nb**”**}.

In[500] **:= $Line1 := Block[{a = ””, c = “Out[“, k = 1}, For[k, k < Infinity, k++, a =**

```
ToString[Out[k]]; If[a == c <> ToString[k] <> "]", Return[k]]]; k]
In[501]:= $Line1
Out[501]= 2014
In[502]:= $Line
Out[502]= 502
```

The above fragment represents source code and examples of application. So, after loading of the user package the values of variables**$Line1** and**$Line** can differ rather significantly**:** the*first* defines total number of the kept**Out–** paragraphs, while the*second–* number of really received**Out–**paragraphs in the current session of the***Mathematica.***

In a whole series of cases of work with large documents there is expediency of deleting from the current session of earlier used**Out**paragraphs with the results unnecessary in the future. This operation is provided by the simple **ClearOut** procedure, whose call**ClearOut[*x*]** returns*nothing* and at the same time deletes**Out**paragraphs with numbers determined by a whole positive number or their list***x.*** The following fragment represents source code of the procedure with a typical example of its application. This procedure in some cases also provides allocation of additional memory in work area of system which in case of large documents is quite significant.

```
In[2520]:= ClearOut[x_ /; PosIntQ[x] || PosIntListQ[x]] :=
Module[{a = Flatten[{x}], k = 1}, Unprotect[Out]; For[k, k <= Length[a], k++,
Out[a[[k]]] =. ]; Protect[Out]; ]
```

```
In[2521] := {Out[1508], Out[1510], Out[1511], Out[1515]}
Out[2521]= {42, 78, 2014, 480}
In[2522]:= ClearOut[{1508, 1510, 1511, 1515}]
In[2523]:= {Out[1508], Out[1510], Out[1511], Out[1515]}
Out[2523]= {%1508, %1510, %1511, %1515}
```

At that, call of used function**PosIntQ[*x*]** or**PosIntListQ[*x*]** returns*True* if***x* –** a positive number or a list positive numbers accordingly**;** otherwise,*False* is returned. These functions are located in our package***AVZ_Package*** [48]**;** at that, many means represented below also use means of this package.

On the other hand, in certain cases of work in the interactive mode a need of replacement of**Out**paragraphs onto other contents arises that rather simple **ReplaceOut** procedure implements, whose successful call**ReplaceOut[*x, y*]** returns*nothing,*at the same time carrying out replacement of contents of the existing**Out–**paragraphs which are determined by a whole positive or their list***x,*** by new expressions defined by a***y*–**argument. The call assumes parity of factual arguments of***x*** and***y;*** otherwise, call**ReplaceOut[*x, y*]** is returned unevaluated. The following fragment represents source code of**ReplaceOut** procedure with typical examples of its usage.

```
In[2025] := AgnAvzVsv = 80
Out[2025]= 80
In[2026]:= ReplaceOut[x_ /; PosIntQ[x] || PosIntListQ[x], y___] :=
```

```
Module[ {a = Flatten[{x}], b = Flatten[{y}], k = 1}, If[b != {}, If[Length[a] !=
Length[b], Defer[ReplaceOut[x, y]], Unprotect[Out]; For[k, k <= Length[a], k++,
Out[a[[k]]] = b[[k]]]; Protect[Out]]; ,
```

ClearOut[x]]] In[2027]**:= ReplaceOut[2025, 480]**
In[2028]**:= Out[2025]**
Out[2028]= 480
In[2029]**:= ReplaceOut[2025]**
In[2030]**:= Out[2025]**
Out[2030]= **%**2025

Moreover, the call**ReplaceOut[*x*]** deletes contents of**Out–**paragraphs that are defined by argument***x*,** generalizing the previous**ClearOut** procedure.

Definition of variables in Mathematica. Like the majority of programming languages in***Mathematica*** system for expressions the names*(identifiers)* are used, giving possibility in the future to address to such named expressions on their names. So, on the operator**"="** the***immediate*** assignment to one or several names of the demanded*expression* is made whereas on the operator ***"x:="–*** the***postponed*** assignment. Distinction of both types of assignment is supposed well known to the reader. For definition of assignment type that has been applied to a name a simple enough procedure**DefOp** can be used whose call**DefOp[*x*]** returns the type in string format of assignment applied to the name***x*** coded also in string format, namely**:*(1)* "*Undefined*" –** a name ***x*** isn't defined**, *(2)* "=" –** the*immediate* assignment has been applied to a name ***x*, *(3)* ":=" –** the*postponed* assignment has been applied to a name***x*.** In[2040]**:= DefOp[x_ /; StringQ[x] && SymbolQ[x] ||**

SymbolQ[ToExpression[x]], y___] := Module[{a = ToString[Definition[x]], b = {y}, c, d}, If[a == "Null", Return["Undefined"], c[h_] := StringTake[a, {Flatten[StringPosition[a, h]][[2]] + 1,–1}]]; If[SuffPref[a, x <> " = ", 1], d = "=", d = ":="]; If[b != {}&& ! HowAct[y], y = c[d]]; d] In[2041]**:= v = 78; g = 66; s := 46; Kr = 18; Art := 25; Res := a + b + c;**

In[2042] **:= Map[DefOp, {"v", "g", "s", "Kr", "Art", "Res", "Avz"}]** Out[2042]= {**"=", "=", ":=", "=", ":=", ":=", "**Undefined"}
In[2043]**:= Clear[y];** {**DefOp["Art", y], y**}
Out[2043]= {**":=", "**25"}
In[2044]**:= Clear[y];** {**DefOp["Res", y], y**}
Out[2044]= {**":=", "**a+ b+ c"}
In[2945]**:= Map[DefOpt,** {**"Kr", "Res"**}**]**
Out[2045]= {"Kr= 18**", "**Res:= a+ b+ c"}

While call **DefOp[*x*, *y*]** through optional second argument***y* –** an undefined variable**–** returns an expression appropriated to a name***x*.** The value which has been assigned to a variable***x*** remains associated with it until its removal on***"x*= . ",** or on the functions**Clear, ClearAll, Remove,** or its redefinition. The fragment above represents source code of the procedure with examples. For evaluation of assignments the***Math*–**language has**Definition** function whose call**Definition[*x*]** returns all definitions ascribed to a name***x*** along with our**DefOpt** procedure*(see fragment above)* which is considered in the present book below. Along with this procedure also other means of return of definitions are considered.

In a number of cases arises a necessity of cleaning of variables of the current session from the values received as a result of dynamic generation. For this purpose it is possible to use the mechanism consisting in accumulation in a list of values of variables which should be removed from the current session subsequently, or be cleared from the values and attributes. For this purpose can be used a function whose call**ClearValues[*w*]** returns the empty list, at the same time deleting all variables having values from the list*w* from the current session**;** whereas the call**ClearValues[*w, y*]** with the second optional argument***y*** –*any expression*– returns the empty list, however such variables are only cleared of values and attributes without removal from the current session. The following fragment represents source code of the**ClearValues** function along with typical examples of its usage.

In[2070]:= **ClearValues[x_ /; ListQ[x], y___] := Select[Map[If[{y}== {}, Remove, ClearAll], Select[Names["`*"], MemberQ[x, ToExpression[#]] &]], # != "Null" &]**

In[2071] := {**a = 42, b = 80, c := 75, d = 480, h5 := 67, Kr = 18, Art = x + Sin[y]**}
Out[2071]= {42, 78, Null, 460, Null, 17, 78+ Sin[2013]}
In[2072]:= **ClearValues[**{**42, 78, 75, 480, 67, 18, x + Sin[y]**}**]**
Out[2072]= {}
In[2073]:= **Names["`*"]**
Out[2073]= {"ClearValues", "Avz", "g", "Res", "s"}
In[2075]:= {**a = 42, b = 78, c := 75, d = 460, h5 := 66, Kr = 17, Art = x + Sin[y]**}
Out[2075]= {42, 75, Null, 450, Null, 16, x+ Sin[y]}
In[2076]:= **ClearValues[**{**42, 78, 75, 460, 66, 17, x + Sin[y]**}**, 78]** Out[2076]= {}
In[2077]:= **Names["`*"]**
Out[2077]= {"a", "Art", "b", "c", "ClearValues", "d", "h5", "Kr"} In[2210]:= **VarsValues[x_ /; ListQ[x]] := Select[Names["`*"],**

MemberQ[x, ToExpression[#]] &] In[2211]:= {**a = 42, b = 78, c := 75, d = 480, h5 := 67, Kr = 18, Art = x + Sin[y]**}: In[2212]:= **VarsValues[**{**42, 78, 75, 480, 67, 18, x + Sin[y]**}**]**
Out[2212]= {"a", "Art", "b", "c", "d", "h5", "Kr"}
In the second part of the fragment the**VarsValues** function is represented, whose call**VarsValues[*x*]** returns the list of variables in string format which have values from a list***x*.** Both functions represent a certain interest during the work in interactive mode of the current session. The recommendations about use of these functions can be found in our book [33].
In some cases on the basis of a certain value is required to determine names to which in the current session this value was ascribed. The given problem is solved by the procedure whose call**Nvalue[*x*]** returns the list of names in string format with a preset value***x*.** At that, the procedure gives only those global variables whose values have been received in the current session in ***In***paragraphs. In the absence of such names the procedure call returns the empty list, i.e. {}. The following fragment presents source code and example of usage of the**Nvalue** procedure.
In[2725]:= **Nvalue[x_] := Module[**{**a =** {}**, b = Names["`*"], k = 1**}**,**

For[k, k <= Length[b], k++, If[ToExpression[b[[k]]] == x, AppendTo[a, b[[k]]], Next[]]]; Select[a, ! SuffPref[#, "Global`", 1] &]

In[2726] := {**Ag, Av, Art, Kr, V, $Ar, Vs, $Kr, G**}**=** {**72, 67, 18, 25, 78, 480, Null, 2014, a*b**}**; Map[Nvalue,** {**72, 67, 18, 25, 78, 480, Null, 2014, a*b**}**]**

Out[2726]= {{"Ag"},{"Av"},{"Art"},{"Kr"},{"V"},{"$Ar"},{"Vs"},{"$Kr"},{"G"}}

The **Nvalue1** procedure is an extension of*functionality* of the above**Nvalue** procedure. The call**Nvalue1[*x*]** returns the list of names of variables in the string format to which in the current session a value***x*** has been ascribed. In the next fragment the source code of**Nvalue1** with examples are presented.

In[4334] **:= Nvalue1[x_] := Module[{a = {}, b = Select[Names["*"], StringFreeQ[#, "$"] &], c, k = 1}, While[k <= Length[b], c = ToExpression["Attributes[" <> ToString1[b[[k]]] <> "]"]; If[! MemberQ[c, Protected], AppendTo[a, b[[k]]], Null]; k++];**

Select[a, ToExpression[#] === x &]] In[4335]**:= {x, y, z, t, h, g, w, s}= {45, 78, 25, 18, 18, 18, 18, 18}; Nvalue1[18]** Out[4335]= {"Art", "g", "h", "s", "t", "u", "w"}
Meanwhile, the**Nvalue1** has not quite satisfactory time characteristics as its algorithm is based on the analysis of all active objects of both the user ones, and the system ones.
For definition of the values ascribed to variables, the procedure**WhatValue** is quite useful whose call**WhatValue[*x*]** returns value ascribed to a variable ***x;*** on an undefined variable***x*** the list of format {"Undefined",***x***} is returned while on a system variable***x*** the list of format {"System",***x***}, and on a local variable***x*** the list of format {"Local",***x***}, is returned. The following fragment represents source code of the**WhatValue** along with examples of its usage. In[2844]**:= WhatValue[x_] := If[SystemQ[x], {"System", x},**

If[! SameQ[Definition2[ToString[x]][[1]], ToString[x]], {"Local", x}, {"Undefined", x}]] In[2845]**:= Ag[x_]:= Module[{}, x^2]; Sv[x_]:= Block[{a}, a+x]; F[x_, y_]:= x*y**
In[2846]**:= Map[WhatValue, {480 + 78*# &, hg, Sin, Ag, Sv, 78, a*b, F, Gs}]**
Out[2846]= {{"Undefined", 480+78#1&}, {"Undefined",hg}, {"System", Sin}, {"Local", Ag}, {"Local", Sv}, {"Undefined", 78}, {"Undefined", a*b}, {"Local", F}, {"Undefined", Gs}}

In[2847] **:= M = Module[{avz}, avz]; WhatValue[M]**
Out[2847]= {"Local", avz$50551}
The call**Clear[*x1, …, xn*]** of the standard function clears symbols {***x1, …, xn***}, excepting symbols with*Protected*-attribute. As a useful generalization of the functions**Clear** and**ClearAll** the procedure**Clear1** can be considered whose call**Clear1[*h,"x1", …, "xn"*]** returns*Null,* i.e. nothing, clearing at condition ***h=1*** the symbols {***x1, x2, …, xn***} with saving of all their attributes and options while at***h= 2,*** clearing symbols {***x1, x2, …, xn***} as from expressions ascribed to them, and from all attributes and options. The fragment below represents source code of**Clear1** along with examples of its usage.

In[2958] **:= Clear1[x_ /; MemberQ[{1, 2}, x], y___] := Module[{a = {y}, b, c, d, k = 1}, If[y === {}, Null, For[k, k <= Length[a], k++, b = a[[k]]; d = Quiet[ToExpression["Attributes[" <> ToString1[b] <> "]"]]; ToExpression["Quiet[ClearAttributes[" <> ToString1[b] <> ", " <> ToString[d] <> "]" <> "; Clear" <> If[x == 1, "", "All"] <> "[" <>**

ToString1[b] <> "]]"]]; If[x == 2, Null, Quiet[Check[ToExpression["SetAttributes[" <> ToString1[b] <> ", " <> ToString[d] <> "]"], $Failed]]]]]] In[2959]**:= S[x_] := x^2; SetAttributes[S, {Listable, Protected}]; Clear["S"];**

Clear **::**wrsym**:** Symbol S is Protected**.** >>

In[2960]**:= Clear1[1, S]**
In[2961]**:= Definition[S]**
Out[2961]= Attributes[S]= {Listable**,** Protected}
In[2962]**:= Clear1[2, S]**
In[2963]**:= Definition[S]**
Out[2963]= Null

As a rather simple and useful tool the **UnDef** procedure serves, whose call **UnDef[*x*]** returns*True* if a symbol***x*** isn**'**t defined, and*False* otherwise. While call**UnDef[*x*, *y*]** with the second optional argument**–***an undefined variable***–** returns**Head1[*x*]** value through***y*,** where**Head1** is an useful generalization of standard function**Head** considered below. At that, in a number of cases of procedural programming the**UnDef** appears as a quite useful tool also. The fragment represents source code of**UnDef** with examples of its usage.

In[2490] **:= UnDef[x_, y___] := Module[**{**a =** {y}**, b = Quiet[Check[Head1[x], True]]**}**, If[a !=** {}**&& ! HowAct[y], y = b]; If[b === "SetDelayed || TagSetDelayed", True, False]]** In[2491]**:= x = 78; y =** {**a, b**}**; z = a + b; Map[UnDef,** {**t, h, x, y, z, 760**}**]**
Out[2491]= {True**,** True**,** False**,** False**,** False**,** False}

In[2492] **:= A[x_ /; UnDef[x]] := Block[**{a}**, a = 480; a]; y := 2014;** {**A[y], A[78]**}
Out[2492]= {A[2014]**,** A[78]}
In[2493]**:= L =** {**a, b, c, d, h, g, p, v, w**}**; Select[L, UnDef[#] &]** Out[2493]= {a**,** b**,** c**,** d**,** p**,** v**,** w}
In[2494]**:= M[x_] := x;** {**UnDef[M, t], t**}
Out[2494]= {False**,** **"**Function**"**}

Right there it is appropriate to note that on examples of **UnDef1, UnDef2, UnDef3–** the**UnDef** procedure modifications**–** basic distinction between procedures of the types**"***Module***"** and**"***Block***"** is illustrated [28-33]. Therefore the type of procedure should be chosen rather*circumspectly*, giving a certain priority to procedures of*Module*-type. In addition, as the*enclosed* procedures the procedures of*Module*–type are used, as a rule.
In a number of cases exists a need of definition of a context of an arbitrary symbol. This problem is solved by a simple enough procedure, whose call **Affiliate[*x*]** returns the context for an arbitrary symbol***x*** given in the string format whereas**"**Undefined**"** is returned on a symbol, completely undefinite for the current session. At that, under**"***completely undefinite***"** is understood as a concrete expression, and a symbol for the first time used in the current session. The fragment below represents source code of the given procedure and examples of its usage, including examples explaining the essence of the concept**"***completely undefinite***".**

In[80]**:= Affiliate[x_ /; StringQ[x]] := Module[**{**a = Quiet[Context[x]]**}**, If[ToString[a] === "Context[" <> x <> "]", "Undefined",**

If[MemberQ[Contexts[], a] && ToString[Quiet[DefFunc[x]]] == "Null" || Attributes[x] === {**Temporary**}**, "Undefined", a]]]** In[81]**:= G = 67; Map[Affiliate,** {**"ProcQ1", "Sin", "G", "Z", "Affiliate"**}**]** Out[81]= {**"**AladjevProcedures`**"**, **"**System`**"**, **"**Global`**"**, **"**Undefined**"**,

" AladjevProcedures`**"**}
In[82]**:=** {**V, G = 72, 67**}**; Map[Affiliate,** {**"V", "G", "80", "Sin[18]", "Q", "Map"**}**]**

Out[82]= {”Undefined“, “Global`”, “Undefined“, “Undefined“, “Undefined“, “System`”}

The call **WhatObj[*x*]** of a quite simple procedure returns value depending on location of a***x***symbol activated in the current session, namely**:*”System”*** a system function**;*”CS”* –** a symbol whose definition has been defined in the current session;***”Undefined”* –** an undefinite symbol**;*”Context’”* –** a context defining a package loaded into the current session and containing definition of***x***symbol**;** if***x*** has a type other than*Symbol,* the procedure call is returned as unevaluated. The following fragment represents source code of**WhatObj** procedure along with examples of its usage.

In[2139] **:= WhatObj[x_ /; SymbolQ[x]]:= Module[**{**a = Quiet[Context[x]], t**}**, If[a === “System`”, “System”, If[a === “Global`”, If[MemberQ[**{**$Failed, “Undefined”**}**, PureDefinition[x]], “Undefined”, “CS”], a]]]**

In[2140]**:= w[x_] := Block[**{}**, x]; Map[WhatObj,** {**Sin, a/b, ProcQ, t78, h6, w**}**]**
Out[2140]= {”System“, WhatObj[a/b], “AladjevProcedures`”, “Undefined“, “Undefined“, “CS”}

For testing of symbols to which expressions are ascribed, ***2*** simple functions **HowAct** and**SymbolQ** are defined. The first of them correctly tests the fact of*definiteness* of a variable in the current session, however on local variables of procedures the call of**HowAct** returns*True* irrespective of existence for them of values. On the other hand, on undefinite local variables of*blocks* the **HowAct** returns*False***.** The call**SymbolQ[*x*]** of simple though rather useful function returns*True* if***x*** is a symbol, and*False* otherwise. Function is used in a number of tools presented in the present book. The following fragment represents source codes of both functions with examples of their usage.

In[2020] **:= HowAct[x_] := If[Quiet[Check[ToString[Definition[x]], True]] === “Null”, False, If[Quiet[ToString[Definition[x]]] === “Attributes[” <> ToString[x] <> “] =** {**Temporary**}**“, False, True]]**

In[2021]**:= SymbolQ[x_] := ! SameQ[Quiet[Check[ToExpression[“Attributes[” <> ToString[x] <> “]”], $Failed]], $Failed]**

In[2022] **:= Map[HowAct,** {**80, IAN, “RANS”, Cos, Args, TestArgsTypes, Label, HowAct, a +b, Agn**}**]**
Out[2022]= {True, False, True, True, True, True, True, True, True, False}
In[2023]**:= Map[SymbolQ,** {**80, IAN, “RANS”, Cos, Args, Label, HowAct**}**]**
Out[2023]= {False, True, True, True, True, True, True, True}
In certain cases the**SymbolQ1** function, being of a modification of function **SymbolQ** can be useful, whose call**SymbolQ1[*x*]** returns*True* if***x*** is a*single* symbol, and*False* otherwise [33]. In [33] certain features of usage of**HowAct** for testing of definiteness of local variables of procedures can be found. In a number of cases exists a need of removal from the current session of a certain active object having the appropriated value with possibility of its subsequent restoration in the current or other session. The given problem is solved by the function whose call**ActRemObj[*x*,*y*]** depending on a value {***“Act”*, *“Rem”***} of the second actual argument deletes an object given by his name in string format from the current session or activates it in the current or other session respectively. The fragment below represents source code of the**ActRemObj** procedure along with examples of its

usage.

In[647] **:=ActRemObj[x_ /; StringQ[x], y_ /; MemberQ[{“Act”, “Rem”}, y]] := Module[{a = $HomeDirectory <> “\” <> x <> “.$ArtKr$”, b, c = ToString[Definition4[x]]}, If[c === “$Failed”, $Failed, If[HowAct[x] && y == “Rem”, b = OpenWrite[a]; WriteString[b, c]; Close[b]; ClearAllAttributes[x]; Remove[x]; “Remove”, If[! HowAct[x] && y == “Act”, If[FileExistsQ[a],**

b = OpenRead[a]; Read[b]; Close[b]; DeleteFile[a]; “Activate”, Return[Defer[ActRemObj[x, y]]]]]]]]] In[648]:= **F := {72, 67, 47, 18, 25}; SetAttributes[F, Protected]; Definition[F]** Out[648]= Attributes[F]= {Protected}

F **:=** {72**,** 67**,** 47**,** 18**,** 25}
In[649]**:= ActRemObj[“F”, “Rem”];**
Out[649]= **“**Remove**”**
In[650]= **Definition[F]**
Out[650]= Null
In[651]**:= ActRemObj[“F”, “Act”];**
Out[651]= **“**Activate**”**
In[652]= **Definition[F]**
Out[652]= Attributes[F]= {Protected}

F **:=** {72**,** 67**,** 47**,** 18**,** 25}
In[653]**:= A[x_] := Module[**{**a=480**}**, x+a]; A[x_, y_] := Module[**{**a=80**}**, x+y+a]**
In[654]**:=** {**A[100], A[100, 200]**}

Out[654] = {580**,** 380}
In[655]**:= ActRemObj[“A”, “Rem”]; Definition[A]**
Out[655]= Null
In[656]**:= ActRemObj[“A”, “Act”];** {**A[100], A[100, 200]**}
Out[656]= {590**,** 380}

Successful removal of an object from the current session returns **“**Remove**”** whereas its restoration in the current session returns**”**Activate**“.** If a datafile containing definition of a removed object***x*,** wasn**’**t found in system catalog **$HomeDirectory,** the call of**ActRemObj** procedure is returned*unevaluated;* on an inadmissible argument***x*** the call**ActRemObj[*x*, *y*]** returns$Failed.

System **Maple** has a rather useful**restart** command which causes the**Maple** kernel to clear its*internal* memory so that system**Maple** acts*almost* as if just started. While the**Mathematica** system has no similar means in interactive mode. The next procedure to a certain extent compensates for this*deficiency*. The call**Restart[]** returns*nothing***,** deleting from the*current session* all objects defined in it. Moreover, from the given list are excluded the objects whose definitions are in the downloaded packages. While the call**Restart[*x*]** with optional argument***x* –** a context or their list defining the user packages that have been loaded in the current session**–** also returns nothing, additionally deleting from the current session all objects whose definitions are contained in the mentioned user packages. The following fragment represents source code of the**Restart** procedure along with examples of its application.

In[2450]**:= Restart[x___] := Module[**{}**, Map[**{**Quiet[ClearAttributes[#, Protected]],**

Quiet[Remove[#]]}&, Names["`*"]]; If[{x}!= {}, Quiet[Map[Remove[# <> "*"] &, Flatten[{x}]]]]]

In[2451] := **F := {72, 67, 47, 18, 25}; SetAttributes[F, Protected]; Sv = 47; a := 6; A[x_] := Module[{a = 480}, x+a]; A[x_, y_] := Module[{a = 80}, x*y*a];**
In[2452]:= **Restart["AladjevProcedures`"]**
In[2453]:= **Map[Definition, {F, A, Map13, HowAct, Sv, a, ActUcontexts}]**
Out[2453]= {Null, Null, Null, Null, Null, Null, Null]}

Moreover, the system objects are not affected by the **Restart.** In a number of cases the function seems a rather useful, allowing to substantially restore an initial state of the current session and to save internal memory of system too.

Means of work with sequential structures. Sequences of expressions*(simply sequences)* in the environment of many languages are formed on the basis of the*comma* operator***","*** and form a certain base for definition of many types of data*(inquiries of procedures,lists,sets,indexes,etc.).* At that, in***Mathematica*** system the given structure as an independent one is absent, and instead of it the list structure protrudes**;** some programming languages adhere to the same concept. In this context a number of simple enough means has been created that ensure operating with the object**Seq[*x*]** defining a sequence of elements***x.*** So, the procedure call**SeqToList[*x*]** provides converting of***Seq–*** object***x*** into the list, the procedure call**ListToSeq[*x*]** provides converting of a list***x*** into***Seq–***object, the procedure call**SeqIns[*x, y, z*]** returns the result of inserting in***Seq–***object***x*** of an arbitrary element***y(list, Seq–object,expression, etc.)*** according to the given position***z(z<= 0 –beforex, z >=*Length[*x*]***–after x,differently–after a**z–position in**x),* the procedure call**SeqToString[*a, b,…*]** returns the list of arguments in string format, whereas the call**SeqUnion[*x, y,…*]** returns result of merge of an arbitrary number of sequences**.** Means for manipulating with***Seq-***objects can be rather widely expanded, providing the user with rather useful program tools. In a certain relation these tools allow to solve the problem of compatibility with other tools, for example, with the ***Maple*** system [28-33].
Meanwhile, the**Mathematica** system provides the function**Sequence[*a,…*]** that defines a sequence of arguments which are automatically transferred to a block, function or module. In this context the call**SequenceQ[*s*]** provides testing of the objects that are created on the basis of the**Sequence** function returning*True* if a***s–***object is defined by this function, and*False* otherwise**;** moreover, the name of***s–***object is coded in string format [33]. On the basis of the standard**Sequence** function it is possible to create quite simple tools ensuring working with sequential structures similarly to the**Maple** system**;** these functions along with the considered ones in [28-33] are rather useful in work with objects of type***"****sequence****",*** whose structure isn**'**t supported by the**Mathematica** and for work with which system has no standard means. The call**Sequence[*x_1, x_2, …, x_n*]** of the standard function defines a sequence of actual arguments***xj(j=1..n),*** transferred to a function. Meanwhile, with objects of type***"****sequence****"*** the**Mathematica** system can work mediately, in particular, on the basis of the list structures. In this regard for expansion of standard**Sequence** function onto list structures the**Sequences** procedure is defined, whose call**Sequences[*x*]** provides insert in a function of arguments ***x*** given by a sequence or a list**;** as a simplified variant of**Sequences** the**Sq** function serves. The following fragment represents source codes of function **Sq** along with the**Sequences** procedure, including their applications.

In[3495]:= **Sequences[x__] := Module[{a = Flatten[{x}], b, c},**
b = "Sequence[" <> ToString[a] <> "]"; a = Flatten[StringPosition[b, {"{", "}"}]];
ToExpression[StringReplace[b, {StringTake[b, {a[[1]], a[[1]]}]–> "", StringTake[b,
{a[[–1]], a[[–1]]}]–> ""}]]]

In[3496] **:= {F[Sequence[{x,y,z}]], F[Sequences[{x,y,z}]], F[Sequences[x,y,z]]}**
Out[3496]= {F[{x, y, z}], F[x, y, z], F[x, y, z]}
In[3497]**:= Sq[x_List] := ToExpression["Sequence[" <>**

StringTake[ToString1[x], {2,–2}] <> "]"] In[3498]**:= Plus[Sq[{72, 66, 56, 47, 25, 18}]]**
Out[3498]= 284
In[3499]**:= G[a, b, c, Sequences[x, y, z]]**
Out[3499]= G[a, b, c, x, y, z]
At work with sequential structures a rather useful is a procedure, providing converting of strings of a special format into lists, and vice versa. The call **ListStrList[*x*]** on a list***x*= {*a, b, …*}** returns a string***s*** of the format*"ahbh…",* while***x*= ListStrList[*s*]** where***h*= FromCharacterCode[*2*].** In case of absence in a***s***string of***h***symbol the call**ListStrList[*s*]** returns the string***s.*** Fragment represents source code of the procedure along with examples its usage. In[2604]**:= ListStrList[x_ /; StringQ[x] || ListQ[x]] :=**

Module[{a = FromCharacterCode[2]}, If[StringQ[x] && ! StringFreeQ[x, a],
Map[ToExpression, StringSplit[x, a]], If[ListQ[x], StringTake[StringJoin[
Mapp[StringJoin, Map[ToString1, x], a]], {1,–2}], x]]]

In[2605] **:= L = ListStrList[{Avz, 72, Agn, 67, Art, 25, Kr, 18, Vsv, 47}]** Out[2605]= **"**Avz 72 Agn 67 Art 25 Kr 18 Vsv 47**"**
In[2606]**:= ListStrList[ListStrList[{Avz, 72, Agn, 67, Art, 25, Kr, 18, Vsv, 47}]]**
Out[2606]= {Avz, 72, Agn, 67, Art, 25, Kr, 18, Vsv, 47}

Chapter 2. Additional means of processing of expressions in the*Mathematica*software

A number of useful means of processing of the expressions supplementing standard means of**Mathematica** system is presented in the present chapter. Analogously to the most software systems the **Mathematica** understands everything with what it manipulates as*"expression"(graphics,lists,formulas, strings,modules,functions,numbers of various type,etc.).* And although all these expressions at first sight rather significantly differ,**Mathematica** represents them in so–called full format. And only the postponed assignment**":="** has no full format. For the purpose of definition of heading*(the type defining it)* of an expression the standard**Head** function is used, whose call**Head[*expr*]** returns the heading of an expression***expr,*** for example**:**

In[6]**:= Map[Head, {Map, Sin, 80, a+b, Function[{x}, x], G[x], S[6], x*y, x^y}]**
Out[6]= {Symbol, Symbol, Integer, Plus, Function, G, S, Times, Power}

For more exact definition of headings we created an useful modification of the standard**Head** function in the form of the**Head1** procedure expanding its opportunities, for example, it concerns testing of operations of*postponed* calculations when on them the values***SetDelayed||TagSetDelayed,*** blocks**,** functions, modules are returned. The call**Head1[*x*]** returns the heading of an expression***x*** in the context {*Block*,*Function*,*Module*,*PureFunction*,*System*, *SetDelayed*||*TagSetDelayed*,*Symbol*,*Head*[*x*]}. The fragment represents source code of the**Head1** procedure with examples of its application comparatively with the**Head** function as it is illustrated by examples of the next fragment, on which functional distinctions of both means are rather evident.

In[2160] **:= Head1[x_] := Module[{a, b, c = Quiet[Check[Attributes[x], {}]]}, If[Quiet[SystemQ[x]], OptRes[Head1, System], If[c != {}, ClearAllAttributes[x]]; b = Quiet[StringSplit[ToString[Definition[x]], "\n \n"]]; a = If[! SameQ[b, {}] && b[[–1]] == "Null", If[Head[x] == Symbol, Symbol, SetDelayed||TagSetDelayed], If[PureFuncQ[x], PureFunction, If[Quiet[Check[FunctionQ[x], False]], Function, If[BlockQ[x], Block, If[BlockModQ[x], Module, Head[x]]]]]]]; {OptRes[Head1, a], Quiet[SetAttributes[x, c]]}[[1]]]]** In[2161]**:= G := S; Z[x_] := Block[{}, x]; F[x_] := x; Map[Head, {ProcQ, Sin, 6, a+b, # &, G, Z, Function[{x}, x], x*y, x^y, F}]**

Out[2161] = {Symbol, Symbol, Integer, Plus, Function, Symbol, Symbol, Function, Times, Power, Symbol}
In[2162]**:= Map[Head1, {ProcQ, Sin, 6, a + b, # &, G, Z, Function[{x}, x], x*y, x^y, F}]**
Out[2162]= {Module, System, Integer, Plus, PureFunction, Symbol, Block, PureFunction, Times, Power, Function}

So, the **Head1** procedure has a quite certain meaning for more exact*(relative to system standard)* classification of expressions according to their headings. On many expressions the calls of**Head1** procedure and**Head** function are identical whereas on a number their calls significantly differ. The concept of expression is the important unifying principle in the system having*identical* internal structure that allows to confine a rather small amount of the basic operations. Meanwhile, despite identical basic structure of expressions, the **Mathematica** system provides a set of various functions for work both with an expression in general, and with its separate components.

Means of testing of correctness of expressions. **Mathematica** has a number of the means providing testing of*correctness* of syntax of expressions among which only two functions are available to the user, namely**:**
SyntaxQ["*x*"]–*returns True,if***x** *–a syntactic correct expression;otherwise False is returned;*
SyntaxLength ["*x*"]–*returns the number***p***of symbols,since the beginning of a string***"x"***that defines syntactic correct expression***StringTake["*x*",{*1*,*p*}];***at that, in case***p**> **StringLength["*x*"]***the system declares that whole string***"x"***is correct, demanding continuation.*
In our opinion, it isn't very conveniently in event of software processing of expressions. Therefore extensions in the form of the**SyntaxQ1** function and **SyntaxLength1** procedure whose source codes along with examples of their application are represented below.

In[2029] **:= SyntaxQ1[x_ /; StringQ[x]] := If[Quiet[ToExpression[x]] === $Failed, False, True]**
In[2030]**:= Map[SyntaxQ1, {"(a+b/", "d[a[1]] + b[2]"}]**
Out[2030]= {False, True}

In[2031]:= **SyntaxLength1[x_ /; StringQ[x], y___] := Module[{a = "", b = 1, d, h = {}, c = StringLength[x]},**

While[b <= c, d = Quiet[ToExpression[a = a <> StringTake[x, {b, b}]]]; If[! SameQ[d, $Failed], h = Append[h, StringTrim[a]]]; b++]; h = DeleteDuplicates[h]; If[{y}!= {}&& ! HowAct[{y}[[1]]], {y}= {h}];

If[h == {}, 0, StringLength[h[[–1]]]]]]

In[2437] := {**SyntaxLength1["d[a[1]] + b[2]", g], g**}
Out[2437]= {14, {"d", "d[a[1]]", "d[a[1]]+ b", "d[a[1]]+ b[2]"}}
In[2438]:= **SyntaxLength1["d[a[1]] + b[2]"]**
Out[2438]= 14

The call **SyntaxLength1[*x*]** returns the maximum number***p*** of position in a string***x*** such that**ToExpression[StringTake [*x*, {*1*, *p*}]]**– a syntactic correct expression**,**otherwise***0*** is returned**;** the call**SyntaxLength1[*x*, *y*]** through the second optional argument***y***–*an undefinite variable*– additionally returns the list of substrings of a string***x*** representing correct expressions.

Means of processing of expressions at the level of their components. Means of this group provide quite effective differentiated processing of*expressions*. The combined symbolical architecture of**Mathematica** gives the possibility of direct generalization of the element-oriented list operations onto arbitrary expressions, supporting operations both on separate terms, and on sets of terms at the given levels in trees of expressions. Without going into details into all means supporting work with components of expressions, we will give only the main from them that have been complemented by our means. Whereas with more detailed description of standard means of this group, including admissible formats of coding, it is possible to get acquainted or in the Help, or in the corresponding literature on the**Mathematica** system, for example, in works [51,52,60,66,71].
The call**Variables[*p*]** of standard function returns the list of all independent variables of a polynomial***p*,** while its application to an arbitrary expression has certain limitations. Meanwhile for receiving all independent variables of an expression***x*** it is quite possible to use quite simple function whose call **UnDefVars[*x*]** returns the list of all independent variables of an expression ***x*.**Unlike the**UnDefVars** function, the call**UnDefVars1[*x*]** returns the list of all independent variables in string format of an expression***x*.** Source codes of both functions with examples of their application are given below in the comparative context with**Variables** function. In some cases the mentioned functions have certain preferences relative to standard**Variables** function.

In[2024] **:= UnDefVars[x_] := Select[OP[x], Quiet[ToString[Definition[#]]] == "Null" &]**
In[2025]:= **UnDefVars[(x^2–y^2)/(Sin[x] +Cos[y]) +a*Log[x+y+z–G[h, t]]]**
Out[2025]= {a, G, h, t, x, y, z}
In[2026]:= **Variables[(x^2–y^2)/(Sin[x] + Cos[y]) + a*Log[x + y +z–G[h, t]]]**
Out[2026]= {a, x, y, Cos[y], Log[x+ y+ z– G[h, t]], Sin[x]}
In[2027]:= **UnDefVars1[x_] := Select[ExtrVarsOfStr[ToString[x], 2], ! SystemQ[#] &]**
In[2028]:= **Map[UnDefVars1, {a+b, a*Sin[x]*Cos[y], {a, b}, a*F[h, g, s]+ H}]**
Out[2028]= {{"a", "b"}, {"a", "x", "y"}, {"a", "b"}, {"a", "F", "g", "h", "s"}}

The call **Replace[*x, r* {,*w*}]** of standard function returns result of application of a rule*r* of the form*a*→*b* or the list of such rules for transformation of an expression*x* as a whole; application of the*3rd* optional argument*w* defines application of rules*r* to parts of*w*–level of an expression*x.* Meanwhile, the standard **Replace** function has a number of restrictions some from which the procedure considerably obviates, whose call**Replace1[*x, r*]** returns the result of application of rules*r* to all or selective independent variables of an expression*x.* In case of detection by the procedure**Replace1** of empty rules the appropriate message is printed with the indication of the list of those rules*r* which were empty, i.e. whose left parts aren't entered into the list of independent variables of an expression*x.* Fragment below represents source code of**Replace1** with examples of its application; at that, comparison with result of application of the**Replace** function on the same expression is given.

In[2052] **:= Replace1[x_, y_ /; ListQ[y] && DeleteDuplicates[Map[Head, y]] == {Rule}|| Head[y] == Rule] := Module[{a = x//FullForm//ToString, b = UnDefVars[x], c, p, l, h = {}, r, k =1, d = ToStringRule[DeleteDuplicates[Flatten[{y}]]]}, p = Mapp[RhsLhs, d, "Lhs"]; c = Select[p, ! MemberQ[Map[ToString, b], #] &]; If[c != {}, Print["Rules " <> ToString[Flatten[Select[d, MemberQ[c, RhsLhs[#, "Lhs"]] &]]] <> " are vacuous"]]; While[k <= Length[d], l = RhsLhs[d[[k]], "Lhs"]; r = RhsLhs[d[[k]], "Rhs"]; h = Append[h, {"[" <> l–> "[" <> r, " " <> l–> " " <> r, l <> "]"–> r <> "]"}]; k++]; Simplify[ToExpression[StringReplace[a, Flatten[h]]]]]**

In[2053] **:= X = (x^2–y^2)/(Sin[x] + Cos[y]) + a*Log[x + y]; Replace[X, {x–> a + b, a–> 80, y–> Cos[a], z–> Log[t]}]**
Out[2053]= a Log[x+ y]+ (x^2– y^2)/(Cos[y]+ Sin[x])
In[2054]:= **Replace1[X, {x–> a+b, a–> 80, y–> Cos[a], z–> Log[t], t–> c+d}];** Rules {z–> (Log[t])**,** t–> (c+ d)} are vacuous
Out[2054]= 80 Log[a+ b+ Cos[a]]+ ((a+ b)^2– Cos[a]^2)/(Cos[Cos[a]]+ Sin[a+ b])
In[2055]:= **Replace1[X, {x–> a + b, a–> 80, y–> Cos[a]}]**
Out[2055]= 80 Log[a+ b+ Cos[a]]+ ((a+ b)^2– Cos[a]^2)/(Cos[Cos[a]]+ Sin[a+ b])

In certain cases at conversions of expressions by means of substitutions the necessity of converting into string format of the left and right parts of rules ***"a→b"*** arises. The given problem is solved by rather simple**ToStringRule** procedure, whose call**ToStringRule[*x*]** returns the rule or the list of rules*x***,** whose left and right parts have string format**;** at that, its right part is taken in parentheses. So, this procedure is used by the above–presented**Replace1** procedure. The procedure **ToStringRule1** is similar to **ToStringRule,** but the right parts of the result is not taken into parentheses. The next fragment represents source code of the**ToStringRule** with examples of its usage.

In[2723] **:=ToStringRule[x_ /; ListQ[x]&& DeleteDuplicates[Map[Head, x]] == {Rule}|| Head[x] == Rule] := Module[{a = Flatten[{x}], b = {}, c, k = 1}, While[k <= Length[a], c = a[[k]]; b = Append[b, ToString[RhsLhs[c, "Lhs"]]–> "(" <> ToString[RhsLhs[c, "Rhs"]] <> ")"]; k++]; If[ListQ[x], b, b[[1]]]]**

In[2724] **:=** {**ToStringRule[a–> b], ToStringRule[**{**a–> b, c–> d, m–> n**}**]**} Out[2724]= {"a"–> "(b)", {"a"–> "(b)", "c"–> "(d)", "m"–> "(n)"}} The call**Level[*x, n*]** of standard function returns list of all subexpressions of an expression*x* at levels from*1* to*n.* As an useful generalization of function is procedure whose call**Levels[*x, h*]** returns the list of all subexpressions for an expression*x* at all its possible levels while through the second argument ***h*** **–** an independent variable– the maximum number of levels of expression ***x*** is returned. Generally speaking, the following defining relation takes place **Levels[*x*, *h*]≡Level[*x*, *Infinity*],** however in case of the**Levels** the procedure additionally returns*maximum* level of an expression***x*.** While the procedure call**ExprOnLevels[*x*]** returns the enclosed list, whose elements are the lists of subexpressions of an expression***x*** which are located on each of its levels from the*first* to the*last.* The fragment below represents source codes of both procedures with examples of their application in a comparative context with the**Level** function with the second*Infinity*–argument.

In[2868]**:= Levels[x_, h_ /; ToString[Definition[h]] == "Null"] := Module[**{**a =** {}**, b, k = 1**}**, While[k < Infinity, b = Level[x, k]; If[a == b, Break[], a = b]; k++]; h = k–1; a]**

In[2869] **:=** {**Levels[(x^2–y^2)/(Sin[x]+Cos[y])+a*Log[x+y+z–G[h, t]], g], g**}
Out[2869]= {{a, x, y, z,–1, h, 7, G[h, 7],–G[h, 7], x+y+z– G[h, 7], Log[x+y+ z– G[h, 7]], a Log[x+ y+ z– G[h, 7]], x, 2, x^2,–1, y, 2, y^2,–y^2, x^2– y^2, y, Cos[y], x, Sin[x], Cos[y]+ Sin[x],–1, 1/(Cos[y]+Sin[x]), (x^2– y^2)/ (Cos[y]+ Sin[x])}, 6}
In[2870]**:= Level[(x^2–y^2)/(Sin[x]+Cos[y])+a*Log[x+y+z–G[h, t]], Infinity]**
Out[2870]= {a, x, y, z,–1, h, 7, G[h, 7],–G[h, 7], x+ y+ z– G[h, 7], Log[x+y+ z– G[h, 7]], a Log[x+ y+ z– G[h, 7]], x, 2, x^2,–1, y, 2, y^2,–y^2, x^2– y^2, y, Cos[y], x, Sin[x], Cos[y]+ Sin[x],–1, 1/(Cos[y]+ Sin[x]), (x^2– y^2)/ (Cos[y]+ Sin[x])}

In[2879]**:= ExprOnLevels[x_] := Module[**{**a =**{}**, k = 1**}**, While[k <= Depth[x], a = Append[a, MinusList[Level[x, k], Level[x, k–1]]]; k++]; a[[1 ;;–2]]]**

In[2880] **:= X = (x^2–y^2)/(Sin[x] + Cos[y]) + a*Log[x + y + z–G[h, t]];** In[2880]**:= ExprOnLevels[X]**
Out[2880]= {{a Log[x+ y+ z– G[h, t]], (x^2– y^2)/(Cos[y]+ Sin[x])}, {a, Log[x+ y+ z– G[h, t]], x^2– y^2, 1/(Cos[y]+ Sin[x])}, {x+ y+ z– G[h, t], x^2,–y^2, Cos[y]+ Sin[x],–1}, {x, y, z, x, 2,–1, y^2, Cos[y], Sin[x],–1}, {G[h, t], y, 2, y, x,–1}, {h, t}}
Relative to the above**Levels** procedure the standard**Depth** function defines on the same expression the maximum number of levels more on***1*,** namely: In[3790]**:= Clear[t];** {**Levels[a + b + c^2, t], t, Depth[a + b + c^2]**} Out[3790]= {{a, b, c, 2, c^2}, 2, 3}
The standard**FreeQ** function provides testing of entries into an expression of subexpressions while a simple **FreeQ1** procedure significantly expands the**FreeQ** function, providing*broader* testing of entries into an expression of subexpressions. The call**FreeQ1[*x*, *y*]** returns*True* if an expression***x*** doesn't contain subexpressions***y*,** otherwise*False* is returned. The**FreeQ2** function expands the**FreeQ** function additionally onto the list as the*2nd* argument. At that, the call**FreeQ2[*x*,*p*]** returns*True* if an expression***x*** doesn't contain subexpression***p*** or subexpressions from a list***p*,** otherwise*False* is returned. The fragment represents source codes of**FreeQ1** and**FreeQ1** with examples of their application in a comparative context with the**FreeQ** function.

In[2202] **:= FreeQ1[x_, y_] := Module[**{**h**}**, Quiet[FreeQ[Subs[x, y, h = Unique["ArtKr"]], h]]]**
In[2203]**:=** {**FreeQ1[a/Sqrt[x], Sqrt[x]], FreeQ[a/Sqrt[x], Sqrt[x]]**}
Out[2203]= {False, True}
In[2204]**:=** {**FreeQ1[**{**Sqrt[x], 18, 25**}**, Sqrt[x]], FreeQ[**{**Sqrt[x], 18, 25**}**, Sqrt[x]]**}
Out[2204]= {False, False}

In[2250]:= **FreeQ2[x_, p_] := If[ListQ[p], If[DeleteDuplicates[Map10[FreeQ, x, p]] === {True}, True, False], FreeQ[x, p]]**
In[2251]:= **L = {a, b, c, d, f, g, h}; {FreeQ[L, {a, d, h}], FreeQ2[L, {a, d, h}]}**
Out[2251]= {True, False}
In[2252]:= **{FreeQ[Cos[x]*Ln[x], {Sin, Ln}], FreeQ2[Cos[x]*Ln[x], {Sin, Ln}]}**
Out[2252]= {True, False}

Using the **FullForm** function providing representation of expressions in the full form can be received a rather useful procedure solving the replacement problem in expressions of the given subexpressions. The call**Replace3[*x,y,z*]** returns the result of replacement in an arbitrary expression***x*** of all entries of subexpressions***y*** into it onto expressions***z;*** as procedure arguments {***y, z***} separate expressions or their lists can be used. At that, in case of arguments {***y,z***} in the form of the list, for them the common length determined by the relation**Min[Map[Length, {*y, z*}]]** is chosen, allowing to avoid the possible especial and erroneous situations, but with the printing of the appropriate diagnostic information as illustrates an example below. The next fragment represents source code of the procedure with examples of its usage.

In[2062] **:= Replace3[x_, y_, z_] := Module[{a = Flatten[{y}], b = Flatten[{z}], c, t = x, k = 1}, c = Min[Map[Length, {a, b}]]; If[c < Length[a], Print["Subexpressions " <> ToString1[a[[c + 1 ;;–1]]] <> " were not replaced"]]; For[k, k <= c, k++, t = Subs[t, a[[k]], b[[k]]]]; t]**

In[2063] **:= Replace3[x^2 + Sqrt[1/a^2 + 1/a–Sin[1/a]], 1/a, Cos[h]]** Out[2063]= x^2+ Sqrt[1/a^2+ Cos[h]– Sin[Cos[h]]]
In[2064]:= **Replace3[1/(1 + 1/a) + Cos[1/a + Sin[1/a]]*(c + 1/a)^2, 1/a, F[h] + d]**
Out[2064]= 1/(1+ d+ F[h])+ Cos[d+ F[h]+ Sin[d+ F[h]]] (c+ d+ F[h])^2 In[2065]:= **Replace3[x^2 + Sqrt[1/a^2 + 1/a–Sin[1/a]], {1/a, 1/b, 1/c}, Cos[h]]**

Subexpressions {b^(–1), c^(–1)} were not replaced
Out[2065]= x^2+ Sqrt[1/a^2+ Cos[h]– Sin[Cos[h]]]

In some cases exists necessity to execute the exchange of values of variables with the corresponding exchange of all them attributes. So, variables***x*** and ***y*** having values***72*** and***67*** should receive values***42*** and***47*** accordingly with the corresponding exchange of all their attributes. The procedure**VarExch** solves the given problem, returning*Null***,** i.e. nothing. The list of two names of variables in string format which exchange by values and attributes or the nested list of*ListList*–type acts as the actual argument**;** anyway all elements of pairs have to be definite, otherwise the call returns*Null* with print of the corresponding diagnostic message.

On the other hand, the call **Rename[*x, y*]** in the regular mode returns*Null*, i.e. nothing, providing replacement of a name***x*** of some defined object onto a name***y*** with preservation of all attributes of this object. At that, the name ***x*** is removed from the current session by means of function**Remove.** But if ***y***–argument defines a name of a defined object or an undefined name with attributes, the procedure call is returned unevaluated. If the first argument ***x*** is illegal for renaming, the procedure call returns*Null***,** i.e. nothing**;** at that, the**Rename** procedure successfully processes also objects of the same name of the type***"Block", "Function"*** or***"Module".*** The**Rename1** procedure is a quite useful

modification of the above procedure, being based on procedure **Definition2** [33]. The call**Rename1[*x, y*]** is similar to the call**Rename[*x, y*]** whereas the call**Rename1[*x, y, z*]** with the third optional***z***–argument–*an arbitrary expression*– performs the same functions as the call**Rename1[*x, y*]** without change of an initial object***x*.**
The**VarExch1** procedure is a version of the above procedure**VarExch** and is based on usage of the**Rename** procedure and global variables**;** it admits the same type of the actual argument, but unlike the second procedure the call**VarExch1[*L*]** in case of detection of undefinite elements of a list***L*** or its sublists is returned*unevaluated* without print of any diagnostic message. In the fragment below, the source codes of the procedures**Rename, Rename1, VarExch** and**VarExch1** along with examples of their usage are represented.

In[2545]:= **VarExch[L_List /; Length[L] == 2|| ListListQ[L] && Length[L[[1]]] == 2] := Module[{Kr, k=1},**

Kr[p_List] := Module[{a = Map[Attributes, p], b, c, m, n}, ToExpression[{"ClearAttributes[" <> StrStr[p[[1]]] <> "," <> ToString[a[[1]]] <> "]", "ClearAttributes[" <> StrStr[p[[2]]] <> "," <> ToString[a[[2]]] <> "]"}] ; {b, c}= ToExpression[{"ToString[Definition[" <> StrStr[p[[1]]] <> "]]", "ToString[Definition[" <> StrStr[p[[2]]] <> "]]"}]; If[MemberQ[{b, c}, "Null"], Print[VarExch::"Both arguments should be defined but uncertainty had been detected: ", p]; Return[], Null]; {m, n}= Map4[StringPosition, Map[StrStr, {b, c}], {" := ", " = "}];

{ n, m}= {StringTake[b, {1, m[[1]][[1]]–1}] <> StringTake[c, {n[[1]][[1]],–1}], StringTake[c, {1, n[[1]][[1]]–1}] <> StringTake[b, {m[[1]][[1]],–1}]}; ToExpression[{n, m}]; Map[ToExpression, {"SetAttributes[" <> StrStr[p[[1]]] <> "," <>

ToString[a[[2]]] <> "]", "SetAttributes[" <> StrStr[p[[2]]] <> "," <> ToString[a[[1]]] <> "]"}]]; If[! ListListQ[L], Kr[L], For[k, k <= Length[L], k++, Kr[L[[k]]]]];]
In[2546]:= **Agn = 67; Avz := 72; Art := 25; Kr = 18; SetAttributes["Agn",**

Protected]; SetAttributes["Art", {Protected, Listable}] In[2547]:= **Map[Attributes, {"Agn", "Avz", "x", "y", "Art", "Kr"}]** Out[2547]= {{Protected}, {}, {}, {}, {Listable, Protected}, {}}
In[2548]:= **VarExch[{{"Avz", "Agn"}, {"x", "y"}, {"Art", "Kr"}}]**

VarExch **::**Both arguments should be defined but uncertainty had been detected**:** {x, y}
In[2549]:= {**Avz, Agn, Art, Kr**}
Out[2549]= {67, 72, 18, 25}
In[2550]:= **Map[Attributes,** {**"Agn", "Avz", "Art", "Kr"**}**]**
Out[2550]= {{}, {Protected}, {}, {Listable, Protected}}

In[2551]:= **Rename[x_String /; HowAct[x], y_ /; ! HowAct[y]] := Module[{a, c, d, b = Flatten[{PureDefinition[x]}]},**

If[! SameQ[b, {$Failed}], a = Attributes[x]; c = ClearAllAttributes[x]; d = StringLength[x]; c = Map[ToString[y] <> StringTake[#, {d + 1,–1}] &, b]; Map[ToExpression, c]; Clear[x]; SetAttributes[y, a]]] In[2552]:= **fm = "Art_Kr"; SetAttributes[fm, {Protected, Listable}];**

```
{ fm, Attributes[fm]}
Out[2552]= {”Art_Kr“, {Listable, Protected}}
In[2553]:= Rename[“fm”, Tampere]
In[2554]:= {Tampere, Attributes[Tampere], fm}
Out[2554]= {”Art_Kr“, {Listable, Protected}, fm}
```

```
In[2557]:= VarExch1[L_List /; Length[L] == 2 || ListListQ[L] && Length[L[[1]]] ==
2] := Module[{Art, k = 1, d},

Art[p_List] := Module[ {a = Quiet[Check[Map[Attributes, p], $Aborted]], b, c, m,
n}, If[a == $Aborted, Return[Defer[VarExch1[L]]], Null]; If[HowAct[$Art$], b =
$Art$; Clear[$Art$]; m = 1, Null]; If[HowAct[$Kr$], c = $Kr$; Clear[$Kr$]; n = 1,
Null];

ToExpression[{“ClearAttributes[” <> StrStr[p[[1]]] <> “,” <> ToString[a[[1]]] <>
“]”, “ClearAttributes[” <> StrStr[p[[2]]] <> “, ” <> ToString[a[[2]]] <> “]”}];
ToExpression[{“Rename[” <> StrStr[p[[1]]] <> “,” <> “$Art$” <> “]”,

“Rename[” <> StrStr[p[[2]]] <> “,” <> “$Kr$” <> “]” }]; ToExpression[“Clear[” <>
StrStr[p[[1]]] <> “,” <> StrStr[p[[2]]] <> “]”]; ToExpression[{“Rename[” <>
StrStr[“$Kr$”] <> “,” <> p[[1]] <> “]”,

“Rename[” <> StrStr[“$Art$”] <> “,” <> p[[2]] <> “]”}]; Map[ToExpression,
{“SetAttributes[” <> StrStr[p[[1]]] <> “,” <> ToString[a[[2]]] <> “]”,
“SetAttributes[” <> StrStr[p[[2]]] <> “,” <> ToString[a[[1]]] <> “]”}]; If[m == 1,
$Art$ = b, Null]; If[n == 1, $Kr$ = c, Null]; ]; If[! ListListQ[L], Art[L], For[k, k <=
Length[L], k++, Art[L[[k]]]]]]
```

```
In[2558] := Agn = 67; Avz := 72; Art := 25; Kr = 18; SetAttributes[“Agn”, Protected];
SetAttributes[“Art”, {Protected, Listable}];
In[2559]:= Map[Attributes, {“Agn”, “Avz”, “Art”, “Kr”}]
Out[2559]= {{Protected}, {}, {Listable, Protected}, {}}
In[2560]:= {$Art$, $Kr$}= {80, 480}; VarExch1[{{“Agn”, “Avz”}, {“x”, “y”},
{“Art”, “Kr”}}]
In[2561]:= {{Agn, Avz, Art, Kr}, Map[Attributes, {“Agn”, “Avz”, “Art”, “Kr”}]}
Out[2561]= {{480, 80, 18, 25}, {{}, {Protected}, {}, {Listable, Protected}}}
In[2562]:= {x, y, $Art$, $Kr$}
Out[2562]= {x, y, $Art$, $Kr$}
```

```
In[2572]:= Rename1[x_String /; HowAct[x], y_ /; ! HowAct[y], z___] :=

Module[ {a = Attributes[x], b = Definition2[x][[1 ;;–2]], c = ToString[y]}, b =
Map[StringReplacePart[#, c, {1, StringLength[x]}] &, b]; ToExpression[b];
ToExpression[“SetAttributes[” <> c <> “, ” <> ToString[a] <> “]”]; If[{z}== {},
ToExpression[“ClearAttributes[” <> x <> “, ” <> ToString[a] <> “]; Remove[” <> x
<> “]”], Null]]
```

```
In[2574] := x := 480; y = 480; SetAttributes[x, {Listable, Protected}] In[2575]:=
Rename1[“x”, Trg42]
In[2576]:= {x, Trg42, Attributes[“Trg42”]}
Out[2576]= {x, 480, {Listable, Protected}}
```

In[2577]**:= Rename1[“y”, Trg47, 80]**
In[2578]**:=** {**y, Trg47, Attributes[“Trg47”]**}
Out[2578]= {480**,** 480**,** {}}

Usage in procedures of global variables, in a number of cases will allow to simplify programming, sometimes, significantly. This mechanism sufficient in detail is considered in [33]. Meantime, the mechanism of global variables in**Mathematica** isn**’**t universal, quite correctly working in case of evaluation of definitions of procedures containing*global* variables in the current session in the***Input*–**paragraph**;** whereas in general case it isn**’**t supported when the loading in the current session of the procedures containing global variables, in particular, from***nb*–**files with the subsequent activation of their contents.

For elimination of a similar situation the procedure has been offered, whose call**NbCallProc[*x*]** reactivates a block, function or module***x*** in the current session, whose definition was in a ***nb*–**file loaded into the current session, with return of*Null***,** i.e. nothing. So, the call**NbCallProc[*x*]** reactivates in the current session all definitions of blocks, functions, modules with the same name***x*** and with various headings. All these definitions have to be loaded previously from some***nb***file into the current session and activated by the function***“Evaluate Notebook”*** of the***GUI*.** The fragment below represents source code of the**NbCallProc** procedure with example of its usage for the above**VarExch1** procedure which uses the global variables.

In[2415] **:= NbCallProc[x_ /; BlockFuncModQ[x]] := Module[**{**a = SubsDel[StringReplace[ToString1[DefFunc[x]], “\n \n”–> “;”], “`” <> ToString[x] <> “`”,** {**“[“, “,”**}**,–1]**}**, Clear[x]; ToExpression[a]]**

In[2416]**:= NbCallProc[VarExch1]**

The performed verification convincingly demonstrates that the **VarExch1** procedure containing the global variables and loaded from the***nb*–**file with subsequent its activation*(by”Evaluate Notebook”)***,** is carried out absolutely correctly and with correct functioning of the mechanism of global variables restoring the values after an exit from the**VarExch1** procedure.**NbCallProc** has a number of rather interesting appendices, above all, if necessary of use of the procedures activated in the***Input*–**paragraph of the current session.

In certain cases before updating of definitions of objects *(procedure***,***function***,** *variable***,***etc.)* it is necessary to check existence for them of*Protected*-attribute that simple function provides**;** the call**ProtectedQ[*x*]** returns*True* if an object ***x*** has*Protected*-attribute, and*False* otherwise. A correct expression can act as argument**;** source code of**ProtectedQ** with examples are presented below.

In[2430]**:= ProtectedQ[x_] := If[MemberQ[Attributes1[x], Protected], True, False]**

In[2431] **:= g = 80; Protect[g]; Map[ProtectedQ,** {**Sin, Protect, AVZ, HowAct, 480, Map, “g”**}**]**
Out[2431]= {True**,** True**,** False**,** False**,** False**,** True**,** True}
The list structure is one of basic in**Mathematica** in even bigger degree, than at**Maple.** And**Maple,** and in even bigger degree of the**Mathematica** have a quite developed set of means of processing of the list structures. One of such important enough tools is the converting of expressions into lists**;** for**Maple** such means has the form***convert(Exp, list)***

whereas**Mathematica** of similar means has no, and procedure**ToList** can be in this quality. The procedure call**ToList[*Exp*]** returns the result of converting of an expression***Exp*** into the list. At that, in case of a string***Exp*** the***Exp*** is converted into the symbol– by–symbol list, in case of a list***Exp*** the list***Exp*** is returned whereas in other cases the converting is based on the standard**Map** function. The following fragment represents source code of the**ToList** with examples of its usage.

In[2370] **:= ToList[expr_] := Module[{a, b, c = {}, d, k = 1, n}, If[StringQ[expr], Characters[expr], If[ListQ[expr], expr, a = ToString[InputForm[Map[b, expr]]]; d = StringSplit[a, ToString[b] <> "["];**

For[k, k <= Length[d], k++, n = d[[k]]; c = Append[c, StringTake[n, {1, Flatten[StringPosition[n, "]"]][[–1]]–1}]]]; ToExpression[c]]]] In[2371]**:= ToList[(a*Sin[x] + g[b])/(c + d) + (d + c)/(Cos[y] + h)]**

Out[2371] = {(c+ d)/(h+ Cos[y]), (g[b]+ a Sin[x])/(c+ d)}
In[2372]**:= ToList["qwertyuiopasdfgh"]**
Out[2372]= {"q", "w", "e", "r", "t", "y", "u", "i", "o", "p", "a", "s", "d", "f", "g", "h"} In[2373]**:= ToList[{a, b, c, d, e, f, g, h, a, v, z, a, g, n, A, r, t, K, r}]** Out[2373]= {a, b, c, d, e, f, g, h, a, v, z, a, g, n, A, r, t, K, r}

Maple has two useful means of manipulation with expressions of the type {*range*,*equation*,*inequality*,*relation*}, whose calls***lhs(Exp)*** and***rhs(Exp)*** return the*left* and the*right* parts of an expression***Exp*** respectively. More precisely, the call***lhs(Exp), rhs(Exp)*** returns the value***op(1,Exp), op(2,Exp)*** respectively. Whereas**Mathematica** has no similar useful means. The given deficiency is compensated by the**RhsLhs** procedure, whose source code with examples of use are given below. The call**RhsLhs[*x*,*y*]** depending on a value {"Rhs", "Lhs"} of the second argument***y*** returns right or left part of an expressions ***x*** respectively relatively to the operator**Head[*x*],** while the call**RhsLhs[*x*,*y*, *t*]** in addition through an undefined variable***t*** returns the operator**Head[*x*]** concerning whom splitting of the expression***x*** onto left and right parts was made. The**RhsLhs** procedure can be a rather easily modified in the light of expansion of the analyzed operators**Head[x].**

In[2700]**:= RhsLhs[x_, y__] := Module[{a = Head[x], b = {x, y}, d, c = {{Greater, ">"}, {GreaterEqual, ">="}, {And, "&&"},**

{ Or, "||"}, {LessEqual, "<="}, {Unequal, "!="}, {Rule, "–>"}, {Less, "<"}, {Plus, {"+", "–"}}, {Power, "^"}, {Equal, "=="}, {Span, ";;"}, {NonCommutativeMultiply, ""}, {Times, {"*", "/"}}}}, If[! MemberQ[Flatten[c], a], Return[Defer[RhsLhs1[x, y]]], d = Level[x, 1]]; If[Length[b] > 2 && ! HowAct[b[[3]]], ToExpression[ToString[b[[3]]] <> " = " <> ToString1[Flatten[Select[c, #[[1]] === a &]]]], Null]; If[b[[2]] == "Lhs", d[[1]], If[b[[2]] == "Rhs", d[[2]], Defer[RhsLhs1[x, y]]]]]**

In[2701] **:= Mapp[RhsLhs, {a > b, a+b, a^b, a*b, a–> b, a <= b, a||b, a && b}, "Rhs"]**
Out[2701]= {b, b, b, b, b, b, b, b}
In[2702]**:=** {**RhsLhs[a || b, "Rhs", w], w**}
Out[2702]= {b, {Or, "||"}}
In[2703]**:=** {**RhsLhs[(a + b)*d -> c, "Lhs", x], x**}
Out[2703]= {(a+ b) d, Rule}

In[2704]:= {**RhsLhs[80 ;; 480, "Rhs", s], s**}
Out[2704]= {480, Span}

Maple has means of testing of expressions for the following types, namely: {**`!`, `*`, `+`, `.`, `::`, `<`, `<=`, `<>`, `=`, `@`, `@@`, `^`, `||`, `and`, `or`, `xor`, `implies`, `not`**}

In **Mathematica** the means of such quite wide range are absent and in this connexion the procedure, whose the call**TwoHandQ[*x*]** returns*True* if an expression***x*** has one of the following types

{"+", ">=", "<=", "&&", "||", "–", "^", "", "<", "==", "!=", ">", "–>"}**

and *False* otherwise, is given below; moreover, if the call**TwoHandQ[*x*, *y*]** returns*True*, through the*2nd* optional argument***y*** *–an undefinite variable–* the type of an expression***x*** is returned. The following fragment represents source code of the**TwoHandQ** procedure along with examples of its usage.

In[2937] **:= TwoHandQ[x__] := Module[{a = ToString[InputForm[{x}[[1]]]], b = {"+", ">=", "<=", "&&", "||", "–", "^", "**", "<", "==", "!=", ">", "–>"}, c, d = {x}}, c = StringPosition[a, b]; If[StringFreeQ[a, "–>"] && StringFreeQ[a, ">="] && Length[c] > 2||Length[c] == 0, False, If[Length[d] > 1 && ! HowAct[d[[2]]] && ! ProtectedQ[d[[2]]], ToExpression[ToString[d[[2]]] <> "=" <> ToString[Head[{x}[[1]]]]], Return[Defer[TwoHandQ[x]]]]; True]]**

In[2938] := {**TwoHandQ[a3 <= w, h], h**}
Out[2938]= {True, LessEqual}
In[2939]:= {**TwoHandQ[a–> b, t], t**}
Out[2939]= {True, Rule}
In[2940]:= {**TwoHandQ[a != b, p], p**}
Out[2940]= {True, Unequal}
In[2941]:= **Clear[z];** {**TwoHandQ[a < b && c, z], z**}
Out[2941]= {True, And}
In[2942]:= **Clear[p];** {**TwoHandQ[a || b + c, p], p**}
Out[2942]= {True, Or}

In **Maple** the type of indexed expressions is defined while in**Mathematica** similar means are absent. For elimination of this drawback we represented a number of procedures eliminating this defect. Among them it is possible to note such procedures as**ArrayInd, Ind, Index, IndexedQ, IndexQ** and **Indices** [30-33,48]. In particular, the call**IndexQ[*x*]** returns*True*, if***x*** – any indexed expression, and*False* otherwise; at that, the argument***x*** is given in string format where under the*indexed* is understood an arbitrary expression whose the*reduced* form completed by the index bracket**"]]".** At that, the call **Indices[*x*]** returns the index component of an indexed expression***x*** given in string format, otherwise the call is returned unevaluated. In the same place rather in details the questions of processing of the indexed expressions are considered. In some cases these means simplify programming. In particular, on the basis of the previous procedures**ToList** and**Ind** the**OP** procedure is programmed whose call**OP[*x*]** returns the list of*atomic* elements composing an expression***x*.** The following fragment represents source code of the**OP** along with typical examples of its usage.

In[2620] **:= OP[exp_] := Module[{a = ToString[InputForm[expr]], b = {}, c, d, k, h},**

If[StringTake[a, {–1,–1}] == "]", a = Flatten[Ind[expr]], a = DeleteDuplicates[Quiet[ToList[expr]]]]; Label[ArtKr]; d = Length[a];

For[k = 1, k <= Length[a], k++, h = a[[k]]; c = Quiet[ToList[h]]; If[MemberQ[DeleteDuplicates[c], $Failed], AppendTo[b, Ind[h]], AppendTo[b, c]]]; a = DeleteDuplicates[Flatten[b]]; If[d == Length[a], Sort[a], b = {}; Goto[ArtKr]]]

In[2621] **:= OP[Sqrt[(a + b)/(c + d)] + Sin[x]*Cos[y]]**
Out[2621]= {–1, 1/2, a, b, c, Cos, d, Sin, x, y}
In[2622]**:= OP[(Log[(a + b)/(c + d)] + Sin[x]*Cos[y])/(G[h, g, t]–w^2)]** Out[2622]= {–1, 2, a, b, c, Cos, d, g, G, h, Log, Sin, t, w, x, y}
In[2623]**:= Map[OP, {{Sin[x]}, G[h, g, t], A[m, p]/G[t, q]}]**
Out[2623]= {{Sin, x}, {g, G, h, t}, {–1, A, G, m, p, q, t}}

In **Mathematica** there is no direct equivalent of***op***–function of**Maple,** but it can be defined within axiomatics of the systems by the next procedure that in a number of cases is rather convenient at programming of appendices:

In[2672]**:= Op[*x*_] := Module[{a, b}, a = {}; If[ListQ[x], a = x, Do[a = Insert[a, Part[x][[b]], –1], {b, Length[x]}]]; a]**

In[2673] **:= Op[Sin[x] + Cos[x]]**
Out[2673]= {Cos[x], Sin[x]}
In[2674]**:= Op[{1, 2, 3, 4, 5, 6, 7, 8, 9}]**
Out[2674]= {1, 2, 3, 4, 5, 6, 7, 8, 9}
In[2675]**:= Op[Sqrt[a + b] + Sin[x]–c/d]**
Out[2675]= {Sqrt[1+ a],–(c/d), Sin[x]}
In[2676]**:= Op[(x + y*Cos[x])/(y + x*Sin[y])]**
Out[2676]= {x+ y Cos[x], 1/(y+ x Sin[y])}
In[2677]**:= Map[Op, {Sin[x], Cos[a + b], 1/(a + b)}]**
Out[2677]= {{x}, {a+ b}, {a+ b,–1}}

It is simple to be convinced that the received results of calls of**Op** procedure are*identical* to similar calls of***op***function in**Maple,** taking into account that **Mathematica** doesn't support structure of type*"sequence"* which is replaced with the list. The**Op** procedure is a rather useful in programming. In a number of appendices the undoubted interest presents a certain analog of**Maple**–процедуры***whattype(x)*** that returns the type of an expression***x*** which is one of basic**Maple**–types. The procedure of the same name acts as a similar analog in**Mathematica** whose call**WhatType[*x*]** returns type of an object***x*** of one of basic types {"Module", "DynamicModule", "Block", "Real", "Complex", "Integer", "Rational", "Times", "Rule", "Power", "Alternatives", "And", "List", "Plus", "Condition", "StringJoin", "UndirectedEdge", **…**}. The following fragment represents source code of the procedure with examples of its application for identification of types of various objects.

In[2869]**:= WhatType[x_ /; StringQ[x]] := Module[{b = t, d, c = $Packages, a = Quiet[Head[ToExpression[x]]]},**

If[a === Symbol, Clear[t]; d = Context[x]; If[d == "Global`", d = Quiet[ProcFuncBlQ[x, t]]; If[d === True, Return[{t, t = b}[[1]]],

Return[{**"Undefined", t = b**}**[[1]]]], If[d == "System`", Return[**{**d, t = b**}**[[1]]], Null]], Return[**{**ToString[a], t = b**}**[[1]]]]; If[Quiet[ProcFuncBlQ[x, t]],**

If[MemberQ[{**"Module", "DynamicModule", "Block"**}**, t], Return[**{**t, t = b**}**[[1]]], t = b; ToString[Quiet[Head[ToExpression[x]]]]], t = b; "Undefined"]]**

In[2870] **:= t = 480; x = 80; y := 42.47; z = a + b; J[x_]:=x; Map[WhatType,** {**"Kr", "x", "y", "z", "ProcQ", "Sin", "F[r]", "WhatType", "J"**}**]**
Out[2870]= {**"**Undefined**"**, **"**Integer**"**, **"**Real**"**, **"**Plus**"**, "Module**"**, **"**System`**"**, **"**F**"**, "Module**"**, **"**Function**"**}
In[2871]**:= Map[WhatType,** {**"a^b", "a**b", "3 + 5*I", "**{**42, 47**}**"**, **"a&&b"**}**]**
Out[2871]= {**"**Power**","**NonCommutativeMultiply**","**Complex**","**List**"**, **"**And**"**}
In[2872]**:= Map[WhatType,** {**"a_/; b", "a <> b", "a <–> b", "a|b"**}**]**
Out[2872]= {**"**Condition**"**, **"**StringJoin**"**, **"**UndirectedEdge**"**, **"**Alternatives**"**}

However, it should be noted that the **WhatType** procedure doesn**'**t support exhaustive testing of types, meantime on its basis it is simple to expand the class of the tested types.

The **ReplaceAll** function of**Mathematica** has very essential restrictions in relation to replacement of subexpressions relatively already of very simple expressions as it is illustrated below. As an alternative for this function can be offered the**Subs** procedure which is functionally equivalent to the above standard**ReplaceAll** function, however which is relieved of a number of its shortcomings. The procedure call**Subs[*x, y, z*]** returns result of substitutions in an expression***x*** of entries of subexpressions***y*** onto expressions***z*.** At that, if***x* –** an arbitrary correct expression, then as the*second* and*third* arguments defining substitutions of the format***y*–>*z*,** an unary substitution or their list coded in the form***y*≡**{***y1*, *y2*,…, *yn***} and***z*≡**{***z1*, *z2*,…, *zn***} appear**,** determining a list of substitutions {***y1* –>*z1*, *y2* –>*z2*,…, *yn* –>*zn***} which are carried out consistently in the order defined at the**Subs** procedure call. The following fragment represents and source code of the**Subs** procedure, and a number of bright examples of its usage on those expressions and with those types of substitutions where the**Subs** procedure surpasses the standard**ReplaceAll** function of**Mathematica.** These examples very clearly illustrate advantages of the**Subs** procedure before the similar system means.

In[2968]**:= Subs[x_, y_, z_] := Module[**{**d, k = 2, subs**}**, subs[m_, n_, p_] := Module[**{**a, b, c, h, t**}**, If[! HowAct[n], m /. n–> p,** {**a, b, c, h**}**=**

First[{**Map[ToString, Map[InputForm,** {**m, n, p, 1/n**}**]]**}**]; t = Simplify[ToExpression[StringReplace[StringReplace[a, b–> "(" <> c <> ")"], h–> "1/" <> "(" <> c <> ")"]]]; If[t === m, m /. n–> p, t]]]; ! ListQ[y] && ! ListQ[z], subs[x, y, z], If[ListQ[y] && ListQ[z] && Length[y] == Length[z], d = subs[x, y[[1]], z[[1]]]; For[k, k <= Length[y], k++, d = subs[d, y[[k]], z[[k]]]]; d, Defer[Subs[x, y, z]]]]]]** In[2969]**:= (c + x^2)/x^2 /. x^2–> a**

Out[2969] = (a+ c)/x^2
In[2970]**:= Subs[(c + x^2)/x^2, x^2, a]**
Out[2970]= (a+ c)/a
In[2971]**:= (c + b^2)/x^2 /. x^2–> Sqrt[z]**
Out[2971]= (b^2+ c)/x^2
In[2972]**:= Subs[(c + b^2)/x^2, x^2, Sqrt[z]]**

Out[2972]= (b^2+ c)/Sqrt[z]
In[2973]:= **(a + x^2)/(b + a/x^2) /. x^2–> Sqrt[a + b]**
Out[2973]= (a+ Sqrt[a+ b])/(b+ a/x^2)

In[2974] **:= Subs[(a + x^2)/(b + a/x^2), x^2, Sqrt[a + b]]**
Out[2974]= (a+ Sqrt[a+ b])/(b+ a/Sqrt[a+ b])
In[2975]:= **(a + x^2)/(b + 1/x^2) /. x^2–> Sqrt[a + b]**
Out[2975]= (a+ Sqrt[a+ b])/(b+ 1/x^2)
In[2976]:= **Subs[(a + x^2)/(b + 1/x^2), x^2, Sqrt[a + b]]**
Out[2976]= (a+ Sqrt[a+ b])/(b+ 1/Sqrt[a+ b])
In[2977]:= **Replace[1/x^2 + 1/y^3, {{x^2–> a + b}, {y^3–> c + d}}]** Out[2977]= {1/x^2+ 1/y^3, 1/x^2+ 1/y^3}
In[2978]:= **Subs[1/x^2 + 1/y^3, {x^2, y^3}, {a + b, c + d}]**
Out[2978]= 1/(a+ b)+ 1/(c+ d)
In[2979]:= **Replace[Sqrt[Sin[1/x^2]+Cos[1/y^3]],{{x^2–>a*b},{y^3–> c*d}}]**
Out[2979]= {Sqrt[Cos[1/y^3]+ Sin[1/x^2]], Sqrt[Cos[1/y^3]+ Sin[1/x^2]]} In[2980]:= **Subs[Sqrt[Sin[1/x^2] + Cos[1/y^3]], {x^2, y^3}, {a*b, c*d}]** Out[2980]= Sqrt[Cos[1/(c d)]+ Sin[1/(a b)]]
In[2981]:= **With[{x = a + c, y = b}, Module[{}, x^2 + y]]**
Out[2981]= b+ (a+ c)^2
In[2982]:= **With[{x^2 = a + c, y = b}, Module[{}, x^2 + y]]**

With **::**lvset**:** Local variable specification {x^2=a+c,y=b} contains**…** Out[2982]= With[{x^2= a+ c, y= b}, Module[{}, x^2+ y]]
In[2983]:= **Subs[Module[{}, x^2 + y], {x, y}, {a + c, b}]**
Out[2983]= b+ (a+ c)^2
In[2984]:= **Subs[Module[{}, x^2 + y], {x^2, y}, {a + c, b}]**
Out[2984]= a+ b+ c
In[2985]:= **Replace[(a + x^2/y^3)/(b + y^3/x^2), {{y^3–> m}, {x^2–> n}}]**
Out[2985]= {(a+ x^2/y^3)/(b+ y^3/x^2), (a+ x^2/y^3)/(b+ y^3/x^2)} In[2986]:= **Subs[(a + x^2/y^3)/(b + y^3/x^2), {y^3, x^2}, {m, n}]** Out[2986]= n (a m+ n)/(m (m+ b n))

In[2987]:= **Df[x_, y_] := Module[{a}, If[! HowAct[y], D[x, y], Simplify[Subs[D[Subs[x, y, a], a], a, y]]]]**

In[2988] **:= Df[(a + x^2)/(b + a/x^2), x^2]**
Out[2988]= (a^2+ 2 a x^2+ b x^4)/(a+ b x^2)^2
In[2989]:= **Df[(x + Sqrt[y])/(y + 2*Sqrt[y])^2, Sqrt[y]]**
Out[2989]= (–4 x– 2 Sqrt[y]+ y)/((2+ Sqrt[y])^3 y^(3/2))
In[2990]:= **D[(x + Sqrt[y])/(y + 2*Sqrt[y])^2, Sqrt[y]]**

General **::**ivar**:** Sqrt[y] is not a valid variable**.** >>
Out[2990]= ∂√**y** ((x+ Sqrt[y])/(y+ 2 Sqrt[y])^2)
In[2991]:= **Df[(x + Sqrt[a + Sqrt[x]])/(d + 2*Sqrt[x])^2, Sqrt[x]]** Out[2991]= ((d+ 2 Sqrt[x])/Sqrt[a+ Sqrt[x]]– 8 (Sqrt[a+ Sqrt[x]]+ x))/(2 (d+

2 Sqrt[x])^3)
In[2992]:= **Df[(x + Sqrt[x + b])/(d + 2*Sqrt[x + b])^2, Sqrt[x + b]]** Out[2992]= (d– 2 (2

x+ Sqrt[b+ x]))/(d+ 2 Sqrt[b+ x])^3

In[2993] **:= ReplaceAll1[x_, y_, z_]:= Module[{a,b,c}, If[! HowAct[y], x /. y–>z, c = If[MemberQ[{Plus, Times, Power}, Head[z]], "(" <> ToString[InputForm[z]] <> ")", ToString[z]]; {a, b}= Map[ToString, Map[InputForm, {x, y}]]; If[StringLength[b] == 1, ReplaceAll[x, y–> z], ToExpression[StringReplace[a, b–> c]]]]]**

In[2994] := {**ReplaceAll[c/x^2+x^2, x^2–> t], ReplaceAll[(1+c/x^2)/(b+x^2), x^2–> t]**}
Out[2994]= {t+ c/x^2, (1+ c/x^2)/(b+ t)}
In[2995]:= {**ReplaceAll1[c/x^2+x^2, x^2, a+b], ReplaceAll1[(1 + c/x^2)/(b + x^2), x^2, c+d]**}
Out[2995]= {a+ b+ c/(a+ b), (1+ c/(c+ d))/(b+ c+ d)}

In[2996]:= **Df1[x_, y_] := Module[{a, b, c = "$$Sart25$$Kr18$$"}, If[! HowAct[y], D[x, y],**

{ a, b}= Map[ToString, Map[InputForm, {x, y}]]; Simplify[ToExpression[StringReplace[ToString[InputForm[D[ToExpression[StringReplace[a, b–> c]], ToExpression[c]]]], c–> b]]]]]

In[2997]:= **Df2[x_, y_] := Module[{a}, If[! HowAct[y], D[x, y], Simplify[ReplaceAll1[D[ReplaceAll1[x, y, a], a], a, y]]]]** In[2998]:= **Df1[(x + Sqrt[a + Sqrt[x]])/(d + 2*Sqrt[x])^2, Sqrt[x]]** Out[2998]= ((d+ 2 Sqrt[x])/Sqrt[a+ Sqrt[x]]– 8 (Sqrt[a+ Sqrt[x]]+ x))/(2(d+

2 Sqrt[x]) ^3)
In[2999]:= **Df2[(x + Sqrt[a + Sqrt[x]])/(d + 2*Sqrt[x])^2, Sqrt[x]]** Out[2999]= ((d+ 2 Sqrt[x])/Sqrt[a+ Sqrt[x]]– 8 (Sqrt[a+ Sqrt[x]]+ x))/(2(d+

2 Sqrt[x])^3)
In[3000]:= **Df2[(a/x^2 + 1/x^2)/(c/x^2 + 1/x^2), 1/x^2]**
Out[3000]=–(((a– c) x^2)/(1+ c)^2)

In[3001] := **Df1[(a/x^2 + 1/x^2)/(c/x^2 + 1/x^2), 1/x^2]**
Out[3001]=–(((a– c) x^2)/(1+ c)^2)
In[3002]:= **Df[(a/x^2 + 1/x^2)/(c/x^2 + 1/x^2), 1/x^2]**
Out[3002]=–((2 (a– c) x^6)/(1+ c x^4)^2)
In[3003]:= **Df2[(a + b)/(Sin[x] + Cos[x]), Sin[x] + Cos[x]]**
Out[3003]=–((a+ b)/(Cos[x]+ Sin[x])^2)
In[3004]:= **Df2[Cos[x]/(Sin[x] + Cos[x]), Cos[x]]**
Out[3004]= Sin[x]/(Cos[x]+ Sin[x])^2

A simple enough example of the previous fragment illustrates application of the**Subs** procedure in realization of the**Df** procedure whose call**Df[*x,y*]** provides differentiation of an expression***x*** on any its subexpression***y*,** and rather significantly expands the standard**D** function**;** the examples illustrate some opportunities of the**Df** procedure. At the end of the above fragment the**ReplaceAll1** procedure functionally equivalent to standard**ReplaceAll** function is presented, which relieves a number of shortages of the second. Then on the basis of the procedures**ReplaceAll1** and**StringReplace** some variants of the**Df** procedure, namely the procedures**Df1** and**Df2** that use a number of useful methods of programming are represented. At the same time, they in some cases are more useful than the**Df** procedure what rather visually illustrate the examples given above. At that, the procedures**Df, Df1** and**Df2** rather significantly expand the standard function**D.** The fragment represents source codes of the above procedures and certain examples of their application where they surpass the standard functions**D, ReplaceAll, Rule** and**With** of the**Mathematica** system.

Receiving of similar expansion as well for the standard **Integrate** function which has rather essential restrictions on usage of arbitrary expressions as integration variables is represented quite natural to us. Two variants of such expansion in the form of the simple procedures**Int** and**Int1** that are based on the previous**Subs** procedure have been proposed for the given purpose, whose source codes and examples of application are represented below.

In[2841]**:= Int[x_, y_] := Module[**{**a**}**, If[! HowAct[y], Integrate[x, y], Simplify[Subs[Integrate[Subs[x, y, a], a], a, y]]]]**

In[2842] **:= Int1[x_, y_] := Module[**{**a**}**, If[! HowAct[y], Integrate[x, y], Simplify[ReplaceAll1[Integrate[ReplaceAll1[x, y, a], a], a, y]]]]** In[2843]**:=** {**Int[Sin[a+1/x^2]+c/x^2, 1/x^2], Int1[Sin[a+1/x^2]+c/x^2, 1/x^2]**} Out[2843]= {–Cos[a+ 1/x^2]+ c Log[1/x^2], c/x^4– Cos[a+ 1/x^2]} In[2844]**:=** {**Int[Sin[n/x^2] + m/x^2, x^2], Int1[Sin[n/x^2] + m/x^2, x^2]**} Out[2844]= {–n CosIntegral[n/x^2]+ m Log[x^2]+ x^2 Sin[n/x^2]**,**

– n CosIntegral[n/x^2]+ m Log[x^2]+ x^2 Sin[n/x^2]} In[2845]**:=**
Int1[(a*x^2+b/x^2)/(c*x^2+d/x^2), x^2]
Out[2845]= (a x^2)/c+ ((b c– a d) ArcTan[(Sqrt[c] x^2)/Sqrt[d]])/(c^(3/2)

Sqrt[d])
In[2846]**:= Integrate[(a*x^2 + b/x^2)/(c*x^2 + d/x^2), x^2]**
Integrate**::**ilim**:** Invalid integration variable or limit(s) in x^2**.** >> Out[2846]= Integrate[(b/x^2+ a*****x^2)/(d/x^2+ c*****x^2)**,** x^2]

Meanwhile, a simple enough **Subs1** function can be considered as a certain extension and complement of the previous **Subs** procedure. The function call**Subs1[*x*,** {***y*, *z***}**]** returns the result of replacement of all occurrences of an subexpression***y*** of an expression***x*** onto an expression***z*;**at that, the function call qua of the second argument allows both the simple***2*–**element list, and the list of***ListList*–**type. The function call**Subs1Q[*x*,*y*]** returns*True* if a call **Subs1[*x*, *y*]** is allowable, and*False* otherwise. The fragment below represents source codes of functions**Subs1** and**Subs1Q** with examples of their usage.

In[2700]**:= Subs1[x_, y_ /; ListQ[y] && Length[y] == 2 || ListListQ[y]] := ToExpression[StringReplace[ToString[FullForm[x]], Map[ToString[FullForm[# [[1]]]]–> ToString[FullForm[#[[2]]]] &, If[ListListQ[y], y,** {**y**}**]]]]** In[2703]**:= Subs1[(a/b + d)/(c/b + h/b),** {{**1/b, t^2**}**,** {**d, 590**}}**]** Out[2703]= (590+ at^2)/(c t^2+ ht^2)
In[2718]**:= Subs1Q[x_, y_] := SameQ[x, Subs1[Subs1[x, y], If[ListListQ[y], Map[Reverse, y], Reverse[y]]]]** In[2719]**:= Subs1Q[(a/b + d)/(c/b + h/b),** {{**1/b, t^2**}**,** {**d, 90**}}**]** Out[2719]**=** True

In[2732] **:= Integrate2[x_, y__] := Module[**{**a, b, d, c = Map[Unique["gs"] &,**

Range[1, Length[{y}]]]}, a = Riffle[{y}, c]; a = If[Length[{y}] == 1, a, Partition[a, 2]]; d = Integrate[Subs1[x, a], Sequences[c]]; {Simplify[Subs1[d, If[ListListQ[a], Map[Reverse, a], Reverse[a]]]], Map[Remove, c]}[[1]]]

In[2733] **:= Integrate2[(a/b + d)/(c/b + h/b), 1/b, d]**
Out[2733]= (d ((2 a)/b+ d Log[1/b]))/(2 (c+ h))
In[2734]**:= Integrate2[x^2*y, x, y]**
Out[2734]= (x^3 y^2)/6
In[2735]**:= Integrate2[1/b, 1/b]**
Out[2735]= 1/(2b^2)
In[2736]**:= Integrate2[(a/b + d)/(c/b + h/t), 1/b, 1/t]**
Out[2736]=–((c t(–2 a b h+ a c t+ 4 b c d t)+ 2 (b h+ c t)(a b h– a c t– 2 b c d t)*****

Log[c/b + h/t])/(4 b^2 c^2 h t^2))
In[2737]**:= Integrate2[Sqrt[a + Sqrt[c + d]*b], Sqrt[c + d]]**
Out[2737]= (2 (a+ bSqrt[c+ d])^(3/2))/(3 b)
In[2738]**:= Integrate2[(a/b + d^2)/(c/b + h/b), 1/b, d^2]**
Out[2738]= (2 a d^2+ b d^4 Log[1/b])/(2 b c+ 2 b h)
In[2739]**:= Integrate2[(a*x^2 + b/x^2)/(c*x^2 + d/x^2), x^2, d/x^2]** Out[2739]= (–c x^4 (4 b c– 2 a d+ a c x^4)+ 2 (d+ c x^4) (2 b c– a d+ a c x^4)*****

Log[(d+ c x^4)/x^2])/(4 c^2 x^4)
In[2743]**:= Diff1[x_, y__] := Module[{a, b, d, c = Map[Unique["gs"] &,**

Range[1, Length[{y}]]]}, a = Riffle[{y}, c]; a = If[Length[{y}] == 1, a, Partition[a, 2]]; d = D[Subs1[x, a], Sequences[c]]; {Simplify[Subs1[d, If[ListListQ[a], Map[Reverse, a], Reverse[a]]]], Map[Remove, c]}[[1]]]

In[2744] **:= Diff1[(a*x^2 + b/x^2)/(c*x^2 + d/x^2), x^2, d/x^2]** Out[2744]= (x^4 (2 b c– a d+ a c x^4))/(d+ c x^4)^3
In[2745]**:= Diff1[c + a/b, c + a/b]**
Out[2745]= 1
In[2746]**:= Diff1[(c + a/b)*Sin[b + 1/x^2], a/b, 1/x^2]**
Out[2746]= Cos[b+ 1/x^2]
In[2747]**:= Diff1[(c + a/b)*Sin[d/c + 1/x^2], 1/c, a/b, 1/x^2]** Out[2747]=–d Sin[d/c+ 1/x^2]
On the basis of the previous**Subs1** function an useful enough procedure has been realized, whose call**Integrate2[*x*, *y*]** provides integrating of an arbitrary expression***x*** on the subexpressions determined by a sequence***y*.** At that, the procedure with the returned result by means of**Simplify** function performs a sequence of algebraic and other transformations and returns the simplest form it finds. The previous fragment presents source code of the**Integrate2** procedure along with typical examples of its usage. While the procedure call **Diff1[*x*, *y*]** that is also realized on the basis of the**Subs1** function returns the differentiation result of an arbitrary expression***x*** on the generalized {***y*,*z*, …**} variables which can be an arbitrary expressions. The result is returned in the *simplified* form on the basis of the**Simplify** function. The previous fragment represents source code of the**Diff1** procedure with examples of its usage.

The represented variants of realization of the tools **Df, Df1, Df2, Diff1, Int, Int1,**

Integrate2, ReplaceAll1, Subs and**Subs1** illustrate various receptions rather useful in a number of problems of programming in the**Mathematica** system and, first of all, in problems of the system character. Moreover, the above means rather essentially extend the appropriate system means.

The next fragment represents the means having both independent value, and a number of useful appendices in programming. Two useful functions used in the subsequent procedures of the fragment preface this fragment. The call**ListRulesQ[*x*]** returns*True* if***x*** is the list of rules of the form***a –>b,*** and*False* otherwise. Whereas the**Map17** function generalizes the standard **Map** function onto case of the list of rules as its*second* actual argument. The call**Map17[*F,* {{*a –>b,c –>d, …*}]** where***F –*** the symbol returns the result of the format {***F*[*a*]–>*F*[*b*],*F*[*c*]–>*F*[*d*], …**} without demanding any additional explanations in view of its transparency.
Whereas the procedure call**Diff[*x, y, z, …*]** returns result of differentiation of an expression***x*** on the generalized variables {***x,y, z, …***} that are arbitrary expressions. The result is returned in the simplified view on the basis of the **Simplify** function. The procedure call**Integral1[*x,y, z,…*]** returns result of integrating of an expression x on the generalized variables {***x, y, z, …***} which are arbitrary expressions. The result is returned in the simplified view on the basis of the standard**Simplify** function. The next fragment represents the source codes of the above means**ListRulesQ, Map17, Diff** and**Integral1** along with typical examples of their application.

In[3321]:= **ListRulesQ[x_List] := DeleteDuplicates[Map[RuleQ[#] &, Flatten[x]]] === {True}**

In[3322] := **ListRulesQ[{a–> b, c–> d, t–> g, w–> v, h}]**
Out[3322]= False
In[3323]:= **ListRulesQ[{a–> b, c–> d, t–> g, w–> v, h–> 90}]** Out[3323]= True

In[3324]:= **Map17[x_, y_ /; RuleQ[y] || ListRulesQ[y]] :=
If[RuleQ[y], Map[x, y], Map[Map[x, #] &, y]]**

In[3325] := **Map17[F, a–> b]**
Out[3325]= F[a]–> F[b]
In[3326]:= **Map17[F, {a–> b, c–> d, t–> g, w–> v, h–> 90}]**
Out[3326]= {F[a]–>F[b], F[c]–>F[d], F[t]–>F[g], F[w]–>F[v], F[h]–>F[90]}

In[3432] := **Diff[x_, y__] := Module[{c ={}, d ={}, b = Length[{y}], t = {}, k = 1, h = x, n = g, a = Map[ToString, Map[InputForm, {y}]]}, Clear[g]; While[k <= b, AppendTo[c, Unique[g]]; AppendTo[d, ToString[c[[k]]]]; AppendTo[t, a[[k]]–> d[[k]]]; h = ToExpression[StringReplace[ToString[h // InputForm], t[[k]]]]; h = D[h, c[[k]]]; h = ReplaceAll[h, Map[ToExpression, Part[t[[k]], 2]–> Part[t[[k]], 1]]]; k++]; g = n; Map[Clear, c]; Simplify[h]]**

In[3433] := **Diff[Sin[x^2]*Cos[1/b^3], 1/b^3, x^2]**
Out[3433]=–Cos[x^2]*Sin[1/b^3]
In[3434]:= **Diff[(a + b)/(c + d), a + b, c + d]**
Out[3434]=–(1/(c+ d)^2)
In[3435]:= **Diff[(a + b) + m/(c + d), a + b, 1/(c + d)]**
Out[3435]= 0

In[3436]:= **Diff[1/Sqrt[a + b]*(a + b) + Tan[Sqrt[a + b] + c], Sqrt[a+b], a+b]**
Out[3436]= (Sec[Sqrt[a+ b]+ c]^2*Tan[Sqrt[a+ b]+ c])/Sqrt[a+ b] In[2257]**:=**
Integral1[x_, y__] := Module[{d = {}, t = {}, k = 1, h = x, n = g,

a = Map[ToString, Map[InputForm, {y}]], c = {}, b = Length[{y}]}, Clear[g]; While[k <= b, AppendTo[c, Unique[g]]; AppendTo[d, ToString[c[[k]]]]; AppendTo[t, a[[k]]–> d[[k]]]; h = ToExpression[StringReplace[ToString[h // InputForm], t[[k]]]]; h = Integrate[h, c[[k]]]; h = ReplaceAll[h, Map[ToExpression, Part[t[[k]], 2]–> Part[t[[k]], 1]]]; k++]; g = n; Map[Clear, c]; Simplify[h]]

In[2258] **:= g = 90; Integral1[Sin[x^2] + Cos[1/b^3], 1/b^3, x^2]**
Out[2258]=–(Cos[x^2]/b^3)+ x^2*Sin[1/b^3]
In[2259]**:= Integral1[Sin[x] + Cos[x], x]**
Out[2259]=–Cos[x]+ Sin[x]
In[2260]**:= Integral1[(Sin[x] + Cos[y])*z, x, y, z]**
Out[2260]= (–(1/2))*z^2*(y*Cos[x]– x*Sin[y])
In[2261]**:= Integral1[(a + b)/(c + d), a + b, c + d]**
Out[2261]= (1/2)*(a+ b)^2*Log[c+ d]
In[2262]**:= Integral1[(a + b) + m/(c + d), a + b, 1/(c + d)]**
Out[2262]= (1/2 (a+ b)^2+ ((a+ b) m)/(c+ d))/(c+ d)
In[2263]**:= Integral1[(a + b)/(c + d), a + b, c + d, c + d]**
Out[2263]= 1/2 (a+ b)^2 (c+ d) (–1+ Log[c+ d])

Thus, the procedures **Diff** and**Integral1** have the certain limitations that at usage demand the corresponding wariness**;** some idea of such restrictions is illustrated by the following very simple example, namely:

In[3322] **:= Diff[(a + b*m)/(c + d*n), a + b, c + d]**
Out[3322]=–(m/((c+ d)^2*n))
In[3323]**:= Integral1[(a + b*m)/(c + d*n), a + b, c + d]**
Out[3323]= ((a+ b)^2*m*Log[c+ d])/(2*n)

With the view of elimination of *these* shortcomings two modifications of the functions**Replace** and**ReplaceAll** in the form of the procedures**Replace4** and**ReplaceAll2** have been created respectively. These procedures expand standard means and allow to code the previous two procedures**Integral1** and**Diff** with wider range of correct appendices in the context of use of the generalized variables of differentiation and integration. The procedure call **Replace4[*x*,*a* –> *b*]** returns the result of application to an expression***x*** of a substitution***a* –> *b*,** when as its left part an arbitrary expression is allowed. At absence in the expression***x*** of occurrences of subexpression***a*** the initial expression***x*** is returned. Unlike previous, the call**ReplaceAll2[*x*, *r*]** returns result of application to an expression***x*** of a rule***r*** or consecutive application of rules from the list***r*;** as the left parts of rules any expressions are allowed. In the following fragment the source codes of the procedures**ReplaceAll2** and**Replace4** along with typical examples of their usage are presented.

In[2445]**:= Replace4[x_, r_ /; RuleQ[r]] := Module[{a, b, c, h},**

{ a, b}= {ToString[x //InputForm], Map[ToString, Map[InputForm, r]]}; c = StringPosition[a, Part[b, 1]]; If[c == {}, x, If[Head[Part[r, 1]] === Plus, h = Map[If[(#

```
[[1]] === 1|| MemberQ[{” “, “(“, “[“, “{”}, StringTake[a, {#[[1]]–1, #[[1]]–1}]]) &&
(#[[2]] === StringLength[a] || MemberQ[{” “, “)”, “]”, “}“, “,”}, StringTake[a, {#
[[2]] + 1, #[[2]] + 1}]]), #] &, c], h = Map[If[(#[[1]] === 1 || !
Quiet[SymbolQ[StringTake[a, {#[[1]]–1, #[[1]]–1}]]]) && (#[[2]] ===
StringLength[a]|| ! Quiet[SymbolQ[StringTake[a, {#[[2]] + 1, #[[2]] + 1}]]]), #] &, c]];
h = Select[h, ! SameQ[#, Null] &]; ToExpression[StringReplacePart[a, “(” <> Part[b,
2] <> “)”, h]]]] In[2446]:= Replace4[(c + d*x)/(c + d + x), c + d–> a + b]

Out[2446] = (c+ d*x)/(a+ b+ x)
In[2447]:= Replace[Sqrt[a + b*x^2*d + c], x^2–> a + b]
Out[2447]= Sqrt[a+ c+ b*d*x^2]
In[2448]:= Replace4[Sqrt[a + b*x^2 *d + c], x^2–> a + b]
Out[2448]= Sqrt[a+ c+ b*(a+ b)*d]

In[2458]:= ReplaceAll2[x_, r_ /; RuleQ[r] || ListRulesQ[r]] := Module[{a = x, k = 1},
If[RuleQ[r], Replace4[x, r], While[k <= Length[r], a = Replace4[a, r[[k]]]; k++]; a]]

In[2459] := ReplaceAll[Sqrt[a + b*x^2*d + c], {x^2–> a + b, a + c–> avz}] Out[2459]=
Sqrt[avz+ b*d*x^2]
In[2460]:= ReplaceAll2[Sqrt[a + b*x^2 *d + c], {x^2–> a + b, a + c–> avz}]
Out[2460]= Sqrt[avz+ b*(a+ b)*d]
In[2461]:= ReplaceAll2[x*y*z, {x–> 42, y–> 90, z–> 500}]
Out[2461]= 1 890 000
In[2462]:= ReplaceAll2[Sin[a + b*x^2*d + c*x^2], x^2–> a + b] Out[2462]= Sin[a+
(a+ b)*c+ b*(a+ b)*d]

In[2488] := Difff[x_, y__] := Module[{a=x, a1, a2, a3, b = Length[{y}], c={}, d, k = 1,
n = g}, Clear[g]; While[k <= b, d = {y}[[k]]; AppendTo[c, Unique[g]]; a1 =
Replace4[a, d–> c[[k]]]; a2 = D[a1, c[[k]]]; a3 = Replace4[a2, c[[k]]–> d]; a = a3;
k++]; g = n; Simplify[a3]] In[2489]:= Difff[(a + b)/(c + d + x), a + b, c + d]

Out[2489] =–(1/(c+ d+ x)^2)
In[2490]:= Difff[(a + b*m)/(c + d*n), a + b, c + d]
Out[2490]= 0
In[2588]:= Integral2[x_, y__] := Module[{a = x, a1, a2, a3, b = Length[{y}],

c = {}, d, k = 1, n = g}, Clear[g]; While[k <= b, d = {y}[[k]]; AppendTo[c, Unique[g]];
a1 = Replace4[a, d–> c[[k]]]; a2 = Integrate[a1, c[[k]]]; a3 = Replace4[a2, c[[k]]–> d];
a = a3; k++]; g = n; Simplify[a3]] In[2589]:= Integral2[(a + b*m)/(c + d*n), a + b, c +
d]

Out[2589] = ((a+ b)*(c+ d)*(a+ b*m))/(c+ d*n)
In[2590]:= Integral2[Sqrt[a + c + b*(a + b)*d], a + c, a + b]
Out[2590]= (2/3)*(a+ b)*(a+ c+ a*b*d+ b^2*d)^(3/2)
In[2591]:= Integral2[Sqrt[a + c + h*g + b*d], c + h, b*d]
Out[2591]= (2/3)*(c+ h)*(a+ c+ b*d+ g*h)^(3/2)
In[2592]:= Integral2[(a + c + h*g + b*d)/(c + h), c + h, b*d]
Out[2592]= (1/2)*b*d*(2*a+ 2*c+ b*d+ 2*g*h)*Log[c+ h]
In[2593]:= Integral2[(c + h*m)/(c + h), c + h*m]
Out[2593]= (c+ h*m)^2/(2*(c+ h))
```

On the basis of the procedure **Replace4** the proceduresDiff** and**Integral1** can be expanded the procedures**Diff** and**Integral1.** The call**Difff[*x,y,z,…*]** returns result of*differentiation* of an arbitrary expression***x*** on the*generalized* variables {***y, z, h, t, …***} that are any expressions. The result is returned in the simplified form on the basis of the standard**Simplify** function. Whereas the call**Integral2[*x,y,z,…*]** returns result of*integration* of an expression***x*** on the generalized variables {***y, z, h, t, …***} that are arbitrary expressions. The result is returned in the simplified form on the basis of the**Simplify** function. In the following fragment the source codes of the above means with examples of application are represented. The means given above are rather useful in many cases of manipulations with algebraic expressions that are based on a system of rules, including their symbolical differentiation and integration on the generalized variables that are arbitrary algebraic expressions. The**SEQ** procedure serves as some analog of the built-in***seq*** function of the same name of the**Maple** system, generating sequences of values. The call **SEQ[*x, y, z*]** returns the list of values***x*[*y*]** where***y*** changes within***z*=*m*;;*n*,** or within***z*=*m*;;*n*;;*p*** with***p*** step**;** at that, values {***m*,*n*, *p***} can accept only positive numerical values**;** at***m*** **<=** ***n*** a value***p*** is considered positive value, otherwise negative. Of examples of the next fragment the principle of formation of the list of values depending on the format of the***3rd*** argument is well visually looked through. In case of zero or negative value of the***3rd*** argument a call **SEQ[*x,y, z*]** is returned unevaluated. The next fragment represents source code of the**SEQ** procedure along with typical examples of its usage.

In[2334]**:= SEQ[x_, y_ /; SymbolQ[y], z_ /; Head[z] == Span] :=**

Module[{a = ToString[z], b = {}, c, d = ToString[y], p}, c = ToExpression[StringSplit[a, " ;; "]]; If[DeleteDuplicates[Map[NumberQ, c]] != {True}|| DeleteDuplicates[Map[Positive, c]] != {True}, Return[Defer[Seq[x, y, z]]], If[Length[c] > 2 && c[[3]] == 0, Return[Defer[Seq[x, y, z]]], If[c[[1]] <= c[[2]], p = 1, p = 2]]]; For[y = c[[1]], If[p == 1, y <= c[[2]], y >= c[[2]]–If[p == 1 && Length[c] == 2 || p == 2 && Length[c] == 2, 0, c[[3]]–1]], If[Length[c] == 2, If[p == 1, y++, y—], If[p == 1, y += c[[3]], y–= c[[3]]]], b = Append[b, x]]; {ToExpression["Clear[" <> d <> "]"], b}[[2]]]

In[2335] **:= SEQ[F[k], k, 18 ;; 25]**
Out[2335]= {F[18]**,** F[19]**,** F[20]**,** F[21]**,** F[22]**,** F[23]**,** F[24]**,** F[25]} In[2336]**:= SEQ[F[t], t, 1 ;; 75 ;; 8]**
Out[2336]= {F[1]**,** F[9]**,** F[17]**,** F[25]**,** F[33]**,** F[41]**,** F[49]**,** F[57]**,** F[65]**,** F[73]} In[2337]**:= SEQ[F[t], t, 100 ;; 95]**
Out[2337]= {F[100]**,** F[99]**,** F[98]**,** F[97]**,** F[96]**,** F[95]}
In[2338]**:= SEQ[F[t], t, 42.71 ;; 80 ;; 6.47]**
Out[2338]= {F[42.71]**,** F[49.18]**,** F[55.65]**,** F[62.12]**,** F[68.59]**,** F[75.06]} In[2339]**:= SEQ[F[k], k, 42 ;; 71 ;;–6]**
Out[2339]= SEQ[F[k]**,** k**,** 42;; 71;;–6]
The call**ExprsInStrQ[*x, y*]** of an useful procedure returns*True* if a string***x*** contains correct expressions, and*False* otherwise. While through the second optional argument***y*** *–an undefinite variable–* a list of expressions that are in ***x*** is returned. The next fragment represents source code of the**ExprsInStrQ** procedure along with typical examples of its usage.

In[2360] **:= ExprsInStrQ[x_ /; StringQ[x], y___] := Module[{a = {}, c = 1, d, j, b = StringLength[x], k = 1}, For[k = c, k <= b, k++, For[j = k, j <= b, j++, d = StringTake[x, {k, j}]; If[! SymbolQ[d] && ! SameQ[Quiet[Check[ToExpression[d], $Failed]], $Failed], a = Append[a, d]]]; c++]; a = Mapp[StringTrim1, Mapp[StringTrim1, Map[StringTrim, a], "+", ""], "–", ""]; a = DeleteDuplicates[Map[StringTrim, Select[a, ! SymbolQ[ToExpression[#]] && ! NumericQ[ToExpression[#]] &]]]; If[a == {}, False, If[{y}!= {}&& ! HowAct[{y}[[1]]], {y}= {a}]; True]]**

In[2361] **:= ExprsInStrQ["a (c + d)–b^2 = Sin[x] h"]**
Out[2361]= True
In[2362]**:=** {**ExprsInStrQ["a (c + d)–b^2 = Sin[x] h", t], t**}
Out[2362]= {True**,** {"a*(c+ d)"**,** "a*(c+ d)– b"**,** "a*(c+ d)– b^2"**,** "(c+ d)"**,**

" (c+ d)– b"**,** "(c+ d)– b^2"**,** "c+ d"**,** "b^2"**,** "Sin[x]"**,** "Sin[x] h"**,** "in[x]"**,** "in[x] h"**,** "n[x]"**,** "n[x] h"}}
In[2363]**:=** {**ExprsInStrQ["n*(a+c)/c ", h1], h1**}
Out[2363]= {True**,** {"n*(a+c)"**,** "n*(a+c)/c"**,** "(a+c)"**,** "(a+c)/c"**,** "a+c"}}

In a whole series of problems of manipulation with expressions, including differentiation and integration on the generalized variables, the question of definition of structure of an expression through subexpressions entering in it including any variables is topical enough. The given problem is solved by the**ExprComp** procedure, whose the call**ExprComp[*x*]** returns the set of all subexpressions composing expression***x*,** whereas the call**ExprComp[*x*, *z*],** where the second optional argument***z*** *–an undefined variable–* through***z*** in addition returns the nested list of subexpressions of an arbitrary expression ***x*** on levels, since the first level. The next fragment represents source code of the**ExprComp** procedure with examples of its use. The code contains means from [48] such as**HowAct, Mapp, StringTrim1** and**SymbolQ.** In[3329]**:= ExprComp[x_, z___] := Module[{a = {x}, b, h = {}, F, q, t = 1},**

F[y_List] := Module[{c = {}, d, p, k, j = 1}, For[j = 1, j <= Length[y], j++, k = 1; While[k < Infinity, p = y[[j]]; a = Quiet[Check[Part[p, k], $Failed]]; If[a === $Failed, Break[], If[! SameQ[a, {}], AppendTo[c, a]]]; k++]]; c]; q = F[a]; While[q != {}, AppendTo[h, q]; q = Flatten[Map[F[{#}] &, q]]]; If[{z}!= {}&& ! HowAct[z], z = Map[Select[#, ! NumberQ[#] &] &, h]]; Sort[Select[DeleteDuplicates[Flatten[h],

Abs[#1] === Abs[#2] &], ! NumberQ[#] &]]]

In[3330] **:= ExprComp[(1/b + Cos[a + Sqrt[c + d]])/(Tan[1/b]–1/c^2)]** Out[3330]= {a**,** 1/b**,** b**,**–(1/c^2)**,** c**,** d**,** Sqrt[c+ d]**,** c+ d**,** a+ Sqrt[c+ d]**,** Cos[a+ Sqrt[c+ d]]**,** 1/b+ Cos[a+ Sqrt[c+ d]]**,** Tan[1/b]**,** 1/(–(1/c^2)+ Tan[1/b])**,**–(1/c^2)+ Tan[1/b]}
In[3331]**:= ExprComp[(1/b + Cos[a + Sqrt[c + d]])/(Tan[1/b]–1/c^2), g]**
Out[3331]= {a**,** 1/b**,** b**,**–(1/c^2)**,** c**,** d**,** Sqrt[c+ d]**,** c+ d**,** a+ Sqrt[c+ d]**,** Cos[a+ Sqrt[c+ d]]**,** 1/b+ Cos[a+ Sqrt[c+ d]]**,** Tan[1/b]**,** 1/(–(1/c^2)+ Tan[1/b])**,**–(1/c^2)+ Tan[1/b]}
In[3332]**:= g**
Out[3332]= {{1/b+ Cos[a+ Sqrt[c+ d]]**,** 1/(–(1/c^2)+ Tan[1/b])}**,** {1/b**,** Cos[a+ Sqrt[c+ d]]**,**–(1/c^2)+ Tan[1/b]}**,**
{b**,** a+ Sqrt[c+ d]**,**–(1/c^2)**,** Tan[1/b]}**,** {a**,** Sqrt[c+ d]**,** 1/c^2**,** 1/b}**,** {c+ d**,** c**,** b}**,** {c**,** d}}

An arbitrary expression can be formed by means of arithmetic operators of types**: Plus** (**'+', '–'**)**, Times** (**'*'**, /)**, Power** (**'^'**)**, Indexed***(indexes)* or**Function** *(function)*. At that, expression***a – b*** has type***"+"*** with operands {***a, –b***}**;** while expression***a***/***b –*** the type***"*"*** with operands {***a, b^(–1)***}; expression***a***^***b*** has type***"^"*** with operands {***a, b***}**;** expression***a[b]*** has the***"Function"*** type while expression**a[[b]]** has the***"Indexed"*** type. In this sense it is possible to use a certain*indicator***Cost** for estimation of complexity of calculation of arbitrary expressions. The**Cost** is defined as a polynomial from variables which are names of the above*three* operators,*Indexed* and*Function* with non–negative integer coefficients. The **Cost** procedure provides calculation of indicator**;** its source code with examples of use represents the following fragment. At creation of the source code the function**Sequences** from [48] has been used.

```
In[2455] := Cost[x_] := Module[{f = {Plus, Times, Power, Indexed, Function}, a =
ToString[InputForm[x]], b = {{"+", "–"}, {"*", "/"}, "^"}, c, d = {}, h, k = 1, j, t},
If[StringFreeQ[a, Flatten[{b, "["}]], 0, c = Map[StringCount[a, #] &, b]; While[k <=
3, h = c[[k]]; If[h != 0, AppendTo[d, {f[[k]], h}]]; k++]; If[Set[b, StringCount[a, "[["]]
> 0, AppendTo[d, {f[[4]], b}]]; t = StringPosition[a, "["]; If[t != {}, t = Map[#[[1]] &,
t]; t = Select[Map[If[StringTake[a, {#–1, #–1}] != "[" && StringTake[a, {# + 1, # +
1}] != "[", #] &, t], ! SameQ[#, Null] &]]; If[t != {}, AppendTo[d, {f[[5]],
Length[t]}]]; b = StringPosition[a, "(–"]; {t, b, h}= {0, Map[#[[1]] &, b],
StringLength[a]}; For[k = 1, k <= Length[b], k++, c = ""; For[j = b[[k]], j <= h, j++,
c = c <> StringTake[a, {j, j}]; If[StringCount[c, "{"] === StringCount[c, "}"], f =
Quiet[Check[ToExpression[c], $Failed]]; If[f === $Failed, Continue[], If
[NumberQ[f], t = t + 1]]; Break[]]]]; d = If[t != 0 && d[[1]][[1]] === Plus, d[[1]][[2]]
= d[[1]][[2]]–t; d, d]; Plus[Sequences[Map[#[[2]]*#[[1]] &, d]]]]] In[2456]:=
Cost[z^(h*n–2) + t^3]

Out[2456] = 3*Plus+ 2*Power+ Times
In[2457]:= Cost[(z^(h*n–2) + t^3)/(x*y + c)]
Out[2457]= 4*Plus+ 2*Power+ 3*Times
In[2458]:= Map[Cost, {42.47, 80*d + p^g, AvzAgnVsv}]
Out[2458]= {0, Plus+ Power+ Times, 0}
In[2459]:= Cost[(z^(h*n[80]–2) + t^3)/(x*y + c[480])]
Out[2459]= 2*Function+ 4*Plus+ 2*Power+ 3*Times
In[2460]:= Cost[(a + Sin[–a + v] + x[b[[–80 ;; 480]]]) // Quiet]
Out[2460]= 2*Function+ Indexed+ 4*Plus
```

The procedure call **Cost[*x*]** returns an indicator***Cost*** of the above format for an arbitrary algebraic expression***x;*** at absence for***x*** of operators is returned zero. At that, the procedure is a rather simply disaggregated relative to the calculation of number of operators**Plus.** Means of the present chapter are rather useful and are located in our package***AVZ_Package*** [48].

Chapter 3. Additional means of processing of symbols and string structures in the*Mathematica*system

Without taking into account the fact that **Mathematica** has rather large set of means for work with string structures, necessity of means that are absent in the system arises. Some of such means are presented in the given chapter**;** among them are available both simple, and more difficult which appeared in the course of programming of problems of different purpose as*additional* functions and procedures simplifying or facilitating the programming. Examples of the present chapter illustrate formalization of*procedures* in the **Mathematica** which reflects its basic elements and principles, allowing by taking into account this material to directly start creation, at the beginning, of simple procedures of different purpose which are based on processing of string structures. Here only the procedures of so–called***"system"*** character intended for processing of string structures are considered which, however, represent also the most direct applied interest for programming of various appendices. Moreover, procedures and functions that have quite foreseeable volume of source code that allows to carry out their rather simple analysis are presented here. Their analysis can serve as rather useful exercise for the reader both who is beginning programming in**Mathematica,** and already having rather serious experience in this direction. Later we will understand under***"system"*** means the actually system means, and our means oriented on mass application. At that, it should be noted that string structures are of special interest not only as basic structures with which the system and the user operate, but also as a base, in particular, of dynamic generation of the objects in**Mathematica,** including procedures and functions. The*mechanism* of such*dynamic generation* is quite simple and rather in details is considered in [28-33], whereas examples of its application can be found in examples of source codes of means of the present book. Below we will present a number of useful means for processing of strings in the**Mathematica** system.

So, the call **SuffPref[*S, s, n*]** provides testing of a string***S*** regarding to begin *(**n** = 1)***,** to finish*(**n** = 2)* by a substring or substrings from the list***s*,** or*(**n** = 3)* be limited from both ends by substrings from***s*.** At establishment of this fact the**SuffPref** procedure returns*True***,** otherwise*False* is returned. While the call**StrStr[*x*]** of simple function provides return of an expression***x*** different from string, in string format, and a double string otherwise. In a number of cases the**StrStr** function is useful enough in work with strings, in particular, with the standard**StringReplace** function. The fragment below represents source codes of the above means along with examples of their application.

In[2510]**:= StrStr[x_] := If[StringQ[x], "\"" <> x <> "\"", ToString[x]]** In[2511]**:= Map[StrStr, {"RANS", a + b, IAN, {72, 67, 47}, F[x,y]}]** Out[2511]= {"\"RANS\"", "a+ b", "IAN", "{72, 67, 47}", "F[x, y]"} In[2512]**:= SuffPref[S_String, s_ /; StringQ[s] || ListQ[s] && DeleteDuplicates[Map[StringQ, s]] == {True}, n_ /; MemberQ[{1, 2, 3}, n]] :=**

Module[{a, b, c, k = 1}, If[StringFreeQ[S, s], False, b = StringLength[S]; c = Flatten[StringPosition[S, s]]; If[n == 3 && c[[1]] == 1 && c[[–1]] == b, True, If[n == 1 && c[[1]] == 1, True,

If[n == 2 && c[[–1]] == b, True, False]]]]]

In[2513] **:= SuffPref["IAN_RANS_RAC_REA_90_500", "90_500", 2]** Out[2513]= True
In[2514]**:= SuffPref["IAN_RANS_RAC_REA", {"IAN_RANS", "IAN_"}, 1]**
Out[2514]= True

In[2515]:= **SuffPref[“IAN_RANS_R_REAIAN”, {“IAN_RANS”, “IAN”}, 3]**
Out[2515]= True

The call **Spos[*x, y, p, d*]** calculates number of position of the first entrance of a symbol*y* into a string*x* to the*left (d=0)* or to the*right (d=1)* from the given position***p.*** If substring***y*** doesn’t enter into string***x*** in the specified direction concerning position***p,*** the call of the**Spos** returns zero. Otherwise, the call **Spos[*x, y, p, dir*]** returns number of a position of the first entrance of***y*** into***x*** string to the left***(dir = 0)*** or to the right***(dir = 1)*** from the given position***p;*** in addition, number of position is counted from the beginning of string***x.*** The **Spos** processes the main erroneous situations, returning on them*False****.*** The following fragment represents source code of the**Spos** with examples of its application. A number of means of***AVZ_Package*** [48] use this procedure.

In[820]:= **Spos[x_String, y_String, p_Integer, d_Integer] := Module[{a, b, c}, If[StringFreeQ[x, y], Return[0],**

If[StringLength[y] > 1 || dir != 0 && dir != 1, Return[False], b = StringLength[x]]]; If[p < 1||p > b, False, If[p == 1 && dir == 0, c = 0, If[p == b && dir == 1, c = 0, If[dir == 0, For[a = p, a >= 1, a–= 1,
If[StringTake[x, {a}] == y, Return[a], c]],

For[a = p, a <= b, a += 1, If[StringTake[x, {a}] == y, Return[a], c]]]]]]; If[a == 0 || a == b + 1, 0, a]]
In[821]:= **Q:= “AV80RAN480IN2014”; {Spos[Q, “A”, 10, 0], Spos[Q, “4”, 3, 1], Spos[Q, “0”, 1, 1], Spos[Q, “Z”, 19, 0], Spos[Q, “W”, 19, 0], Spos[Q, “P”, 1, 1]}**

Out[821]= {6, 8, 4, 0, 0, 0}

In a number of cases the possibilities of the standard functions **Replace** and **StringReplace** are insufficient. In this connection the procedure, whose call **StringReplace2[*S, s, E*]** returns the result of replacement of all entries into a string***S*** of its substrings***s*** onto an expression***E*** has been created**;** at that, the replaced substrings***s*** shouldn’t be limited by letters. If the string***S*** doesn’t contain occurrences of***s,*** the procedure call returns the initial string***S*** while on the empty string***S*** the empty string is returned. In a sense the procedure **StringReplace2** combines possibilities of the above system functions. The following fragment represents source code of the**StringReplace2** procedure along with typical examples of its usage.

In[2267]:= **StringReplace2[S_ /; StringQ[S], s_ /; StringQ[s], Exp_] := Module[{b, c, d, k = 1, a = Join[CharacterRange[“A”, “Z”],**

CharacterRange[“a”, “z”]] }, b = Quiet[Select[StringPosition[S, s], ! MemberQ[a, StringTake[S, {#[[1]]–1, #[[1]]–1}]] && ! MemberQ[a, StringTake[S, {#[[2]] + 1, #[[2]] + 1}]] &]]; StringReplacePart[S, ToString[Exp], b]]

In[2268] := **StringReplace2[“Length["abSin[x]"] + Sin[x] + ab–Sin[x]*6”, “Sin[x]”, “a^b”]**
Out[2268]= **“**Length["abSin[x]"]+ a^b+ ab– a^b*6**”**
In[2269]:= **StringReplace2[“Length["abSin[x]"] + Cos[x] + ab–Cos[x]*6”, “abSin[x], “a^b”]**
Out[2269]= **“**Length["a^b"]+ Cos[x]+ ab– Cos[x]*6**”**

In addition to the standard**StringReplace** function and the**StringReplace2** procedure in a number of cases the procedure**StringReplace1** is provided as useful. The call**StringReplace1[*S, L, P*]** returns result of substitution in a string***S*** of substrings from the list***P*** instead of its substrings determined by positions of the nested list***L*** of***ListList***–type. The next fragment represents source code of the**StringReplace1** procedure with examples of its usage.

In[2331]**:= StringReplace1[S_ /; StringQ[S], L_ /; ListListQ[L] && Length[L[[1]]] == 2 && MatrixQ[L, IntegerQ] && Sort[Map[Min, L]][[1]] >= 1, P_ /; ListQ[P]] := Module[**{**a =** {}**, b, k = 1**}**,**

If[Sort[Map[Max, L]][[–1]] <= StringLength[S] && Length[P] == Length[L], Null, Return[Defer[StringReplace1[S, L, P]]]]; For[k, k <= Length[L], k++, b = L[[k]]; a = Append[a, StringTake[S, {**b[[1]], b[[2]]**}**]–> ToString[P[[k]]]]]; StringReplace[S, a]]**
In[2332]**:= StringReplace1["avz123456789agn",** {{**4, 7**}**,** {**8, 10**}**,** {**11, 12**}}**,**

{ **" RANS ", Tampere, Sqrt[(a + b)*(c + d)]**}**]**
Out[2332]= **"**avz RANS TampereSqrt[(a+ b) (c+ d)]agn**"**
For operating with strings the**SubsDel** procedure represents a quite certain interest whose call**SubsDel[*S, x, y, p*]** returns result of removal from a string ***S*** of all substrings which are limited on the right*(at the left)* by a substring***x*** and at the left*(on the right)* by the first met symbol in string format from the list***y;*** moreover, search of***y***symbol is done to the left***(p = –1)*** or to the right ***(p = 1).*** In addition, the deleted substrings will contain a substring***x*** since one end and the first symbol met from***y*** since other end. Moreover, if in the course of search the symbols from the list***y*** weren't found until end of the string***S,*** the rest of the initial string***S*** is removed. The fragment represents source code of the**SubsDel** procedure with examples of its use. Procedure is used by a number of means from our***AVZ_Package*** package [48]. In[2321]**:= SubsDel[S_ /; StringQ[S], x_ /; StringQ[x], y_ /; ListQ[y] &&**

DeleteDuplicates[Map[StringQ, y]] == {**True**}**&& Plus[Sequences[Map[StringLength, y]]] == Length[y], p_ /; MemberQ[**{**–1, 1**}**, p]] := Module[**{**b, c = x, d, h = StringLength[S], k**}**, If[StringFreeQ[S, x], Return[S], b = StringPosition[S, x][[1]]]; For[k = If[p == 1, b[[2]] + 1, b[[1]]–1], If[p == 1, k <= h, k >= 1], If[p == 1, k++, k—], d = StringTake[S,** {**k, k**}**]; If[MemberQ[y, d] || If[p == 1, k == 1, k == h], Break[],**

If[p == 1, c = c <> d, c = d <> c]; Continue[]]]; StringReplace[S, c–> ""]]

In[2322] **:= SubsDel["12345avz6789", "avz",** {**"8"**}**, 1]**
Out[2322]= **"**1234589**"**
In[2323]**:= SubsDel["12345avz6789", "avz",** {**"8", 9**}**, 1]**
Out[2323]= SubsDel["12345avz6789", "avz", {"8", 9}, 1]
In[2324]**:= SubsDel["123456789avz6789", "avz",** {**"5"**}**, 1]**
Out[2324]= **"**123456789**"**

While the procedure call **SubDelStr[*x, L*]** provides removal from a string***x*** of all substrings which are limited by numbers of the positions given by the list***L*** of***ListList–***type from two–element sublists. On incorrect tuples of the actual arguments the procedure call is returned unevaluated. The following fragment represents source code of the procedure with examples of its use.

In[2826]:= **SubDelStr[x_ /; StringQ[x], L_ /; ListListQ[L]] := Module[{k = 1, a = {}}, If[! L == Select[L, ListQ[#] && Length[#] == 2 &] || L[[–1]][[2]] > StringLength[x] || L[[1]][[1]] < 1,**

Return[Defer[SubDelStr[x, L]]], For[k, k <= Length[L], k++, a = Append[a, StringTake[x, L[[k]]]–> ""]]; StringReplace[x, a]]]

In[2827] := **SubDelStr["123456789abcdfdh", {{3, 5}, {7, 8}, {10, 12}}]** Out[2827]= "1269dfdh"
In[2828]:= **SubDelStr["123456789abcdfdh", {{3, 5}, {7, 8}, {10, 12}, {40, 42}}]**
Out[2828]= SubDelStr["123456789abcdfdh", {{3, 5}, {7, 8}, {10, 12}, {40, 42}}]

For receiving of substrings of a string which are given by their positions of end and beginning,**Mathematica** possesses the**StringTake** function having ***6*** formats. However, in a number of cases is more convenient a receiving the sublines limited not by positions, but the list of substrings. For this purpose two functionally identical procedures**StringTake1** and**StringTake2** serve [48]. The call**StringTake**{**1**|**2**}**[*x*,*y*]** returns the list of substrings of a string***x*** that are limited by their substrings***y;*** as the second argument can be both an expression, and their list. The following fragment represents source code of the**StringTake2** procedure along with typical examples of its usage.

In[2751]:= **StringTake2[x_ /; StringQ[x], y_] := Module[{b = {}, k = 1, a = Map[ToString, Map[InputForm, y]]},**

For[k, k <= Length[a], k++, b = Append[b, ToString1[a[[k]]] <> "–>" <> "",""]]; StringSplit[StringReplace[x, ToExpression[b]], ","]]

In[2752] := **StringTake2["ransianavzagnvsvartkr", {ian, agn, art}]** Out[2752]= {"rans", "avz", "vsv", "kr"}
In[2753]:= **StringTake2["ransianavzagnvsvartkr", {ian, 480, art, 80}]** Out[2753]= {"rans", "avzagnvsv", "kr"}
In[2754]:= **StringTake2["ransianavzagnvsvartkr", {ran, ian, agn, art, kr}]**
Out[2754]= {"s", "avz", "vsv"}

For work with strings the following procedure is rather useful, whose call **InsertN[*S*, *L*, *n*]** returns result of inserting into a string***S*** after its positions from a list***n*** of substrings from a list***L;*** in case***n***= {< ***1***|≥ **StringLength[*S*]**} a substring is located before string***S*** or in its end respectively. It is supposed that the actual arguments***L*** and***n*** may contain various number of elements, in this case the excess elements***n*** are ignored. At that, processing of a string ***S*** is carried out concerning the list of positions for insertions***m*** determined according to the following relation***m*** =**DeleteDuplicates[Sort[*n*]].** The call with inadmissible arguments is returned unevaluated. The next fragment represents source code of the**InsertN** procedure with examples of its usage.

In[2583] := **InsertN[S_String, L_ /; ListQ[L], n_ /; ListQ[n] && Length[n] == Length[Select[n, IntegerQ[#] &]]] := Module[{a = Map[ToString, L], d = Characters[S], p, b, k = 1, c = FromCharacterCode[2], m = DeleteDuplicates[Sort[n]]}, b = Map[c <> ToString[#] &, Range[1, Length[d]]]; b = Riffle[d, b]; p = Min[Length[a], Length[m]]; While[k <= p, If[m[[k]] < 1,**

PrependTo[b, a[[k]]], If[m[[k]] > Length[d], AppendTo[b, a[[k]]], b = ReplaceAll[b, c <> ToString[m[[k]]]–> a[[k]]]]]; k++]; StringJoin[Select[b, ! SuffPref[#, c, 1] &]]]

In[2584] := **InsertN["123456789Rans_Ian", {Ag, Vs, Art, Kr}, {6, 9, 3, 0, 3, 17}]**
Out[2584]= "Ag123Vs456Art789KrRans_Ian"
In[2585]:= **InsertN["123456789", {a, b, c, d, e, f, g, h, n, m}, {4, 2, 3, 0, 17, 9, 18}]**
Out[2585]= "a12b3c4d56789efg"

Contrary to the previous procedure the procedure **DelSubStr[*S*, *L*]** provides removal from a string***S*** of substrings, whose positions are given by the list ***L;*** the list***L*** has nesting***0*** or***1,*** for example, {{3, 4}, {7}, {9}} or {1, 3, 5, 7, 9} [48]. Earlier it was already noted that certain functional facilities of**Mathematica** need to be reworked both for purpose of expansion of scope of application, and elimination of shortcomings. It to the full extent concerns such a rather important function as**ToString[*x*]** that returns the result of converting of an arbitrary expression***x*** into*string* format. This standard procedure incorrectly converts expressions into string format, that contain string subexpressions if to code them in the standard way. By this reason we defined**ToString1[*x*]** procedure returning result of correct converting of an arbitrary expression***x*** into string format. The next fragment presents source code of the**ToString1** procedure with examples of its application. In a number of appendices this procedure is popular enough.
In[2720]:= **ToString1[x_] := Module[{a = "$Art25Kr18$.txt", b = "", c, k = 1},**

Write[a, x]; Close[a]; For[k, k < Infinity, k++, c = Read[a, String]; If[SameQ[c, EndOfFile], Return[DeleteFile[Close[a]]; b], b = b <> StrDelEnds[c, " ", 1]]]]]

In[2721] := **Kr[x_] := Module[{a = "ArtKr", b = " = "}, a <> b <> ToString[x]]**
In[2722]:= **ToString[Definition[Kr]]**
Out[2722]= "Kr[x_] := Module[{a= ArtKr, b= = }, a<>b<>ToString[x]]" In[2723]:= **ToExpression[%]**

ToExpression ::sntx: Invalid syntax in or before"Kr[x_]:= Module[{a= …". Out[2723]= $Failed
In[2724]:= **ToString1[Definition[Kr]]**
Out[2724]= "Kr[x_]:= Module[{a= "Art_Kr", b= " = "},

StringJoin[a, b, ToString[x]]]"
In[2725]:= **ToExpression[%]; Kr[80]**
Out[2725]= "Art_Kr= 80"
In[2748]:= **ToString2[x_] := Module[{a}, If[ListQ[x], SetAttributes[ToString1, Listable]; a = ToString1[x]; ClearAttributes[ToString1, Listable]; a, ToString1[x]]]**

In[2749] := **ToString2[a + b/72–Sin[480.80]]**
Out[2749]= "0.13590214677436363+ a+ b/72"
In[2750]:= **ToString2[{{72, 67}, {47, {a, b, {x, y}, c}, 52}, {25, 18}}]** Out[2750]= {{"72", "67"}, {"47", {"a", "b", {"x", "y"}, "c"}, "52"}, {"25", "18"}}

Immediate application of the **ToString1** procedure allows to simplify rather significantly the programming of a number of problems. At that, examples of the previous fragment visually illustrate application of both means on the concrete example, emphasizing advantages of our procedure. Whereas the**ToString2** procedure expands the previous procedure onto lists of any level of nesting. So, the call**ToString2[*x*]** on an argument***x*,** different from the list, is equivalent to the call**ToString1[*x*],** while on a list***x*** is equivalent to the call**ToString1[*x*]** that is endowed with*Listable*–attribute. Source code of the**ToString2** with examples of its usage ended the given fragment.

The next fragment represents rather useful procedure, whose call **SubStr[*S*, *p*,*a*, *b*, *r*]** returns a substring of a string**S** which is limited at the left by the first symbol other than symbol***a*** or other than symbols from the list***a*,** and on the right is limited by symbol other than***b*** or other than symbols from a list***b*.** Meanwhile, through argument***r*** in case of a erroneous situation the corresponding message diagnosing the arisen error situation is returned. A value of argument***p*** must be in interval***1*.. StringLength[*S*].** The following fragment represents source code and examples of usage of this procedure.

In[2379] **:= SubStr[S_/; StringQ[S], p_/; IntegerQ[p], a_ /; CharacterQ[a] || ListQ[a] && DeleteDuplicates[Map[CharacterQ, a]] == {True}, b_ /; CharacterQ[b] || ListQ[b] && DeleteDuplicates[Map[CharacterQ, b]] == {True}, r_ /; ! HowAct[r]] := Module[{c = Quiet[StringTake[S, {p, p}]], k, t}, If[p >= 1 && p <= StringLength[S],**

For[k = p + 1, k <= StringLength[S], k++, t = StringTake[S, {k, k}];

If[If[CharacterQ[b], t != b, ! MemberQ[b, t]], c = c <> t; Continue[], Break[]]]; For[k = p–1, k >= 1, k—, t = StringTake[S, {k, k}]; If[If[CharacterQ[a], t != a, ! MemberQ[a, t]], c = t <> c; Continue[], Break[]]]; c, r = "Argument p should be in range 1.." <> ToString[StringLength[S]] <> " but received " <> ToString[p]; $Failed]]

In[2380] **:= SubStr["12345abcd480e80fg6789sewrt", 14, "3", "r", Error]** Out[2380]= **"**45abcd480e80fg6789sew**"**
In[2382]**:= SubStr["12345abcdefg6789sewrt", 25, "0", "x", Error]** Out[2382]= $Failed
In[2383]**:= Error**
Out[2383]= **"**Argument p should be in range 1**..**21 but received 25**"** In[2384]**:= SubStr["12345ab3c480def80gr6789sewrt", 7, "3", "r", Err]** Out[2384]= **"**45ab3c480def80g"

In a number of cases of processing of expressions the problem of excretion of one or the other type of expressions from strings is quite topical. In this relation a certain interest the procedure**ExprOfStr** represents whose source code with examples of its usage represents the following fragment. The call **ExprOfStr[*w*, *n*, *m*, *L*]** returns result of extraction from a string***w*** limited by its***nth*** position and the end, of the first correct expression on condition that search is done*on the left **(m=–1)**/on the right **(m=1)*** from the given position, furthermore a symbol, next or previous behind the found expression must belong to the list***L*.** The call is returned in string format**;** in the absence of a correct expression$Failed is returned, while procedure call on inadmissible arguments is returned unevaluated.

In[2675]**:= ExprOfStr[x_ /; StringQ[x], n_ /; IntegerQ[n] && n > 0, m_ /; MemberQ[{–1, 1}, m], L_ /; ListQ[L]] :=**

Module[{a = "", b, k}, If[n >= StringLength[x], Return[Defer[ExprOfStr[x, n, m, L]]], Null]; For[k = n, If[m ==–1, k >= 1, k <= StringLength[x]], If[m ==–1, k—, k++], If[m ==–1, a = StringTake[x, {k, k}] <> a, a = a <> StringTake[x, {k, k}]]; b =

Quiet[ToExpression[a]]; If[b === $Failed, Null, If[If[m ==–1, k == 1, k ==StringLength[x]]|| MemberQ[L, Quiet[StringTake[x, If[m ==–1, {k–1, k–1}, {k + 1, k + 1}]]]], Return[a], Null]]]; $Failed]

In[2676] := **P[x_, y_] := Module[{a, P1}, P1[z_, h_] := Module[{n}, z^2 + h^2]; x*y + P1[x, y]]**
In[2677]**:= x = ToString1[Definition[P]];** {**ExprOfStr[x, 44, 1,** {**" ", ";", ","**}**], ExprOfStr[x, 39,–1,** {**" ", ";", ","**}**]**}
Out[2677]= {**"**Module[{n}, z^2+ h^2]**"**, **"**P1[z_, h_]**"**}
In[2679]**:= ExprOfStr[x, 10, 1,** {**" ", ";", ","**}**]**
Out[2679]= $Failed
In[2680]**:= ExprOfStr["12345678;F[(a+b)/(c+d)]; AV_2014", 10, 1,** {**"^", ";"**}**]**
Out[2680]= **"**F[(a+ b)/(c+ d)]**"**

The **ExprOfStr1** procedure represents an useful enough modification of the previous procedure**;** its call**ExprOfStr1[*x*,*n*,*p*]** returns a substring of a string ***x*,** that is minimum on length and in which a boundary element is a symbol in***n*–*th*** position of string***x*,** containing a correct expression. At that, search of such substring is done from***n*–*th*** position to the right and until the end of string***x(p=1)*,** and from the left from***nth*** position of string to the beginning of the string**(*p=–1)*.** In case of lack of such substring the call returns**$**Failed while on inadmissible arguments the call is returned unevaluated [48]. In[3148]**:= x = "123**{**a+b**}**, F[c+d+Sin[a+b]]"; ExprOfStr1[x, 25,–1]** Out[3148]= **"**F[c+d+Sin[a+b]]**"**
In[3149]**:= x = "123**{**a+b**}**, [c+d]"; ExprOfStr1[x, 15,–1]**
Out[3149]= $Failed
In[3150]**:= x = "123**{**a+b**}**, [c+d]"; ExprOfStr1[x, 17,–1]**
Out[3150]= ExprOfStr1[**"**123{a+b}**,** [c+d]**"**, 17**,–**1]

In a certain relation to the **ExprOfStr** procedure also the**ExtrExpr** procedure adjoins, whose call**ExtrExpr[*S*, *N*, *M*]** returns a correct expression in string format which is contained in a substring of a string***S*** limited by positions with numbers***N*** and***M*.** At lack of a correct expression the empty list, i.e. {} is returned [48].
In[2622]**:= ExtrExpr["z=(Sin[x+y] + Log[x])+G[x,y];", 4, 13]**
Out[2622]= **"**Sin[x+ y]**"**
In[2623]**:= ExtrExpr["z=(Sin[x+y] + Log[x])+F[x,y];", 1, 21]**

Out[2623] = **"**Log[x]+ Sin[x+ y]**"**
In[2624]**:= ExtrExpr["z = (Sin[x + y] + Log[x]) + F[x, y];", 1, 36]** Out[2624]= **"**F[x, y]+ Log[x]+ Sin[x+ y]**"**

The **ExtrExpr** procedure is rather useful in a number of appendices, which are connected, first of all, with extraction of expressions from strings. The string structure is one of*basic* structures in**Maple** and in**Mathematica,** for ensuring work with which both systems have a number of the effective enough means. However, if**Maple** along with a rather small set of the built- in means has an expanded set of means from the**StringTools** module and a number of means from our library [47],**Mathematica** in this regard has less representative set of means. Meanwhile, the set of its standard means allows to program enough simply the lacking**Maple–**analogs, and other means of processing of strings. Our means of this orientation are represented in [48].

Unlike the **StringFreeQ** function, the procedure call**StringDependQ[*x*, *y*]** returns*True***,** if a string***x*** contains entries of a substring***y*** or all substrings given by the list***y*,** and*False* otherwise. Whereas the call**StringDependQ[*x*, *y*, *z*]** in the presence of the third optional argument*–an undefinite variable–* throu it in addition returns the list of substrings that don**'**t have entries into a string***x*.** The following fragment represents source code of the procedure **StringDependQ** along with typical examples of its usage.

```
In[2611] := StringDependQ[x_ /; StringQ[x], y_ /; StringQ[y]||ListStrQ[y], z___] :=
Module[{a = Map[StringFreeQ[x, #] &, Flatten[{y}]], b = {}, c = Length[y], k = 1},
If[DeleteDuplicates[a] == {False}, True, If[{z}!= {}&& ! HowAct[z], z =
Select[Flatten[y], StringFreeQ[x, #] &]]; False]]
```

```
In[2612] := {StringDependQ["abcd", {"a", "d", "g", "s", "h", "t", "w"}, t], t}
Out[2612]= {False, {"g", "s", "h", "t", "w"}}
In[2613]:= {StringDependQ["abgschtdw", {"a", "d", "g", "s", "h", "t", "w"}, j], j}
Out[2613]= {True, j}
```

In a number of tasks of strings processing, there is a need of replacement not simply of substrings but substrings limited by the given substrings. The procedure solves one of such tasks, its call**StringReplaceS[*S*,*s1*, *s2*]** returns the result of*substitution* into a string***S*** instead of entries into it of substrings ***s1*** limited by strings***"x"*** on the left and on the right from the specified lists ***L*** and***R*** respectively, by substrings***s2(*****StringLength[*"x"*]=** ***1);*** at absence of such entries the procedure call returns***S*.** The following fragment represents source code of the**StringReplaceS** procedure with an example of its usage.

```
In[2691]:= StringReplaceS[S_ /; StringQ[S], s1_String, s2_String] :=

Module[ {a = StringLength[S], b = StringPosition[S, s1], c = {}, k = 1, p, L =
Characters["`!@#%^&*(){}:"\/|<>?~–=+[];:'., 1234567890"], R =
Characters["`!@#%^&*(){}:"\/|<>?~=[];:'., "]}, If[b == {}, S, While[k <= Length[b],
p = b[[k]]; If[Quiet[(p[[1]] == 1 && p[[2]] == a) || (p[[1]] == 1 && MemberQ[R,
StringTake[S, {p[[2]] + 1, p[[2]] + 1}]])|| (MemberQ[L, StringTake[S, {p[[1]]–1,
p[[1]]–1}]] && MemberQ[R, StringTake[S, {p[[2]] + 1, p[[2]] + 1}]])|| (p[[2]] == a
&& MemberQ[L, StringTake[S, {p[[1]]–1, p[[1]]–1}]])], c = Append[c, p]]; k++];
StringReplacePart[S, s2, c]]]
```

```
In[2692]:= StringReplaceS["abc& c + bd6abc–abc78*abc", "abc", "xyz"] Out[2692]=
"xyz& c+ bd6xyz– abc78*xyz"
```

The given procedure, in particular, is a rather useful means at processing of definitions of blocks and modules in respect of operating with their formal arguments and local variables.

In a number of cases at processing of strings it is necessary to extract from them the substrings limited by the symbol {"}, i.e.*"strings in strings"***.** This problem is solved by the procedure, whose call**StrFromStr[*x*]** returns the list of such substrings that are in a string***x*;** otherwise, the call**StrFromStr[*x*]** returns the empty list, i.e. {}. The following fragment represents source code of the procedure along with typical examples of its application.

```
In[3050] := StrFromStr[x_ /; StringQ[x]] := Module[{a = """, b, c = {}, k = 1}, b =
```

DeleteDuplicates[Flatten[StringPosition[x, a]]]; For[k, k <= Length[b]–1, k++, c = Append[c, ToExpression[StringTake[x, {b[[k]], b[[k + 1]]}]]]; k = k + 1]; c]

In[3051] **:= StrFromStr[“12345"678abc"xyz"48080"mnph”]** Out[3051]= {”678abc“**,** “910”}
In[3052]**:= StrFromStr[“123456789”]**
Out[3052]= {}
Unlike the standard**StringSplit** function, the call**StringSplit1[*x,y*]** performs semantic splitting of a string***x*** by a symbol***y*** onto elements of the returned list. In this case the semantics is reduced to the point that in the returned list only those substrings of the string***x*** which contain correct expressions are placed**;** for lack of such substrings the procedure call returns the empty list. The**StringSplit1** procedure appears as a quite useful means, in particular, at programming of means of processing of headings of blocks, functions and modules. The comparative analysis of**StringSplit** and**StringSplit1** speaks well for the last. The next fragment represents source code of the procedure **StringSplit1** along with typical examples of its application.

In[2950]**:= StringSplit1[x_ /; StringQ[x], y_ /; StringQ[y] || StringLength[y] == 1] :=**

Module[{a = StringSplit[x, y], b, c = {}, d, p, k = 1, j = 1}, d = Length[a]; Label[G]; For[k = j, k <= d, k++, p = a[[k]]; If[! SameQ[Quiet[ToExpression[p]], $Failed], AppendTo[c, p], b = a[[k]]; For[j = k, j <= d–1, j++, b = b <> y <> a[[j + 1]]; If[! SameQ[Quiet[ToExpression[b]], $Failed], AppendTo[c, b]; Goto[G], Null]]]]; c]

In[2951] **:= StringSplit[“x_String, y_Integer, z_/; MemberQ[{1,2,3,4,5}, z]|| IntegerQ[z], h_, s_String, c_ /; StringQ[c] || StringLength[c] == 1”, “,”]**
Out[2951]= {”x_String“**,** “y_Integer“**,** “z_/;MemberQ[{1“**,**”2“**,**”3“**,**” 4“**,**”5}“**,** “z] ||IntegerQ[z]“**,** “ h_”**,** “ s_String“**,** “ s_ /; StringQ[y]||StringLength[y]== 1”}
In[2952]**:= StringSplit1[“x_String, y_Integer, z_/; MemberQ[{1,2,3,4,5}, z]|| IntegerQ[z], h_, s_String, c_ /; StringQ[c] || StringLength[c] == 1”, “,”]**
Out[2952]= {”x_String“**,** “ y_Integer“**,** “ z_/; MemberQ[{1**,** 2**,** 3**,** 4**,** 5}**,** z]|| IntegerQ[z]“**,** “ h_”**,** “s_String“**,** “h_ /; StringQ[y]|| StringLength[y]== 1”}

A number of the problems dealing with processing of strings do the**SubsStr** procedure as a rather useful, whose call**SubsStr[*x, y, h, t*]** returns result of replacement in a string***x*** of all entries of substrings formed by*concatenation (on the right at**t=1**or at the left at**t=0**)* of substrings***y*** with strings from a list ***h*,** onto strings from the list***h*** respectively. At impossibility of carrying out replacement the initial string***x*** is returned. The**SubsStr** procedure appears as a useful means, for example, at programming of means of processing of the body of procedure in string format that contains*local variables*. Whereas the call**SubsBstr[*S, x, y*]** returns the list of all nonintersecting substrings in a string***S*** that are limited by symbols***x*** and***y*,** otherwise the empty list, i.e. {} is returned. The following fragment represents source codes of procedures **SubsStr** and**SubsBstr** along with examples of their usage.

In[2209]**:= SubsStr[x_ /; StringQ[x], y_ /; StringQ[y], h_ /; ListQ[h], t_ /; MemberQ[{0, 1}, t]] := Module[{a = Map[ToString, h], b},**

If[StringFreeQ[x, y], Return[x], b = If[t == 1, Map3[StringJoin, y, a],

Mapp[StringJoin, a, y]]]; If[StringFreeQ[x, b], Return[x], StringReplace[x, Map9[Rule, b, h]]]]

In[2210] := **SubsStr["Module[{a$ = $CallProc, b$, c$}, x + StringLength[y] + b$*c$; b$–c$; a$]", "$", {",", "]", "[", "}", " ", ";", "*", "^", "–"}, 1]**
Out[2210]= "Module[{a= $CallProc,b,c},x+StringLength[y]+b*c; b–c; a]"

In[2438]:= **SubsBstr[S_ /; StringQ[S], x_ /; CharacterQ[x], y_ /; CharacterQ[y]] := Module[{a = {}, c, h, n, m, s = S, p, t}, c[s_, p_, t_] := DeleteDuplicates[Map10[StringFreeQ, s,{p, t}]] == {False};**

While[c[s, x, y], n = StringPosition[s, x, 1][[1]][[1]]; s = StringTake[s, {n,–1}]; m = StringPosition[s, y, 1]; If[m == {}, Return[], m = m[[1]][[1]]]; AppendTo[a, h = StringTake[s, {1, m}]]; s = StringReplace[s, h–> ""]; Continue[]]; a]

In[2439] := **SubsBstr["123452333562675243655", "2", "5"]**
Out[2439]= {"2345", "23335", "2675", "24365"}
In[2440]:= **SubsBstr["123452333562675243655", "9", "5"]**
Out[2440]= {}

The following procedure **SubStrSymbolParity** presents undoubted interest at processing of definitions of the blocks/functions/modules given in string format. The call**SubStrSymbolParity[*x, y, z, d*]** with four arguments returns the list of substrings of a string***x*** that are limited by one-character strings***y, z(y ≠ z);*** at that, search of such substrings in the string***x*** is done from***left to right(d=0),*** and from right to left***(d=1).*** While call**SubStrSymbolParity[*x, y, z, d, t*]** with the fifth optional argument***–****a positive number****t>0 –*** provides search in substring of***x*** that is limited by a position***t*** and the end of string***x*** at***d=0,*** and by the beginning of string***x*** and***t*** at***d=1.*** In case of receiving of inadmissible arguments the procedure call is returned unevaluated, while at impossibility of extraction of the demanded substrings the procedure call returns$Failed. This procedure is a rather useful means, in particular, at the solution of problems of extraction in definitions of procedures of the list of local variables, headings of procedures, etc. The fragment represents source code of the**SubStrSymbolParity** procedure with examples of its application

In[2533]:= **SubStrSymbolParity[x_ /; StringQ[x], y_ /; CharacterQ[y], z_ /; CharacterQ[z], d_ /; MemberQ[{0, 1}, d], t___ /; t == {}|| PosIntQ[{t}[[1]]]] :=**

Module[{a, b = {}, c = {y, z}, k = 1, j, f, m = 1, n = 0, p, h}, If[{t}== {}, f = x, f = StringTake[x, If[d == 0, {t, StringLength[x]}, {1, t}]]]; If[Map10[StringFreeQ, f, c] != {False, False}|| y == z, Return[], a = StringPosition[f, If[d == 0, c[[1]], c[[2]]]]]; For[k, k <= Length[a], k++, j = If[d == 0, a[[k]][[1]] + 1, a[[k]][[2]]–1]; h = If[d == 0, y, z]; While[m != n, p = Quiet[Check[StringTake[f, {j, j}], Return[$Failed]]]; If[p == y, If[d == 0, m++, n++];

If[d == 0, h = h <> p, h = p <> h], If[p == z, If[d == 0, n++, m++]; If[d == 0, h = h <> p, h = p <> h], If[d == 0, h = h <> p, h = p <> h]]]; If[d == 0, j++, j—]];

AppendTo[b, h]; m = 1; n = 0; h = ""]; b]

In[2534] := **SubStrSymbolParity["12345{abcdfgh}67{rans}8{ian}9", "{", "}", 0]**
Out[1534]= {"{abcdfgh}", "{rans}", "{ian}"}

In[2535]:= **SubStrSymbolParity["12345{abcdfg}67{rans}8{ian}9", "{", "}", 0, 7]**
Out[2535]= {"{rans}", "{ian}"}
In[2536]:= **SubStrSymbolParity["12345{abcdfgh}67{rans}8{ian}9", "{", "}", 1]**
Out[2536]= {"{abcdfgh}", "{rans}", "{ian}"}
In[2537]:= **SubStrSymbolParity["12345{abfgh}67{rans}8{ian}9", "{", "}", 1, 25]**
Out[2537]= {"{abfgh}", "{rans}"}
In[2538]:= **SubStrSymbolParity["12345{abch}67{rans}8{ian}9", "{", "}", 1,–80]**
Out[2538]= SubStrSymbolParity["12345{abch}67{rans}8{ian}9", "{", "}",1,–80]
Meanwhile, in many cases it is quite possible to use a simpler and reactive version of this procedure, whose call**SubStrSymbolParity1[*x, y, z*]** with***3*** factual arguments returns the list of substrings of a string***x*** that are limited by one-character strings {***y,z***}***(y≠z);*** at that, search of such substrings is done from left to right. In the absence of the desired substrings the procedure call returns the empty list, i.e. {}. The following fragment represents source code of the**SubStrSymbolParity1** procedure along with examples of its usage.

In[2023] := **SubStrSymbolParity1[x_ /; StringQ[x], y_ /; CharacterQ[y], z_ /; CharacterQ[z]] := Module[{c = {}, d, k = 1, j, p, a = DeleteDuplicates[Flatten[StringPosition[x, y]]], b = DeleteDuplicates[Flatten[StringPosition[x, z]]]},**

If[a == {}|| b == {}, {}, For[k, k <= Length[a], k++, p = StringTake[x, {a[[k]], a[[k]]}]; For[j = a[[k]] + 1, j <= StringLength[x], j++, p = p <> StringTake[x, {j, j}]; If[StringCount[p, y] == StringCount[p, z], AppendTo[c, p]; Break[]]]]; c]]

In[2024] := **SubStrSymbolParity1["Definition2[Function[{x, y}, x*Sin[y]]", "{", "}"]**
Out[2024]= {"{x, y}"}
In[2025]:= **SubStrSymbolParity1["G[x_String, y_, z_/; ListQ[z]]:= Block[{}, {x,y,z}]", "[", "]"]**
Out[2025]= {"[x_String, y_, z_/; ListQ[z]]", "[z]", "[{}, {x, y, z}]"}

The following simple enough procedure is a very useful modification of the **SubStrSymbolParity1** procedure; its call**StrSymbParity[*S, s, x, y*]** returns a list, whose elements are substrings of a string***S*** that have format***sw*** format on condition of parity of the minimum number of entries into a substring***w*** of symbols***x,y (x≠y).*** In the absence of such substrings or identity of symbols ***x,y,*** the call returns the empty list, i.e. {}. The following fragment represents source code of the**StrSymbParity** procedure with examples of its usage.

In[2175]:= **StrSymbParity[S_ /; StringQ[S], S1_ /; StringQ[S1], x_ /; StringQ[x] && StringLength[x] == 1, y_ /; StringQ[y] && StringLength[y] == 1] :=**

Module[{b = {}, c = S1, d, k = 1, j, a = StringPosition[S, S1]}, If[x == y ||a == {}, {}, For[k, k <= Length[a], k++, For[j = a[[k]][[2]] + 1, j <= StringLength[S], j++, c = c <> StringTake[S, {j, j}]; If[StringCount[c, x] != 0 && StringCount[c, y] != 0 && StringCount[c, x] === StringCount[c, y], AppendTo[b, c]; c = S1; Break[]]]]]; b]

In[2176] := **StrSymbParity["12345[678]9[abcd]", "34", "[", "]"]**
Out[2176]= {"345[678]"}
In[2177]:= **StrSymbParity["12345[6[78]9", "34", "[", "]"]**
Out[2177]= {}

In[2178]**:= StrSymbParity["12345[678]9[ab34cd[x]34[a, b]", "34", "[", "]"]**
Out[2178]= {"345[678]"**,** "34cd[x]"**,** "34[a, b]"}

Procedures **SubStrSymbolParity, SubStrSymbolParity1 & StrSymbParity** are rather useful tools, in particular, at processing of definitions of modules and blocks given in string format. These procedures are used by a number of means of the***AVZ_Package*** package [48].

The **SubsStrLim** procedure presents a quite certain interest for a number of appendices which rather significantly use procedure of extraction from the strings of substrings of a quite certain format. The next fragment represents source code of the**SubsStrLim** procedure along with examples of its usage.

In[2542] **:= SubsStrLim[x_ /; StringQ[x], y_ /; StringQ[y] && StringLength[y] == 1, z_ /; StringQ[z] && StringLength[z] == 1] := Module[**{**a, b =x <> FromCharacterCode[6], c =y, d =**{}**, p, j, k = 1, n, h**}**, If[! StringFreeQ[b, y] && ! StringFreeQ[b, z], a =StringPosition[b, y];**

n = Length[a]; For[k, k <= n, k++, p = a[[k]][[1]]; j = p; While[h=Quiet[StringTake[b,{**j+1, j+1**}**]]; h != z, c=c <>h; j++]; c=c <>z; If[StringFreeQ[StringTake[c,** {**2,–2**}**],** {**y, z**}**], AppendTo[d, c]]; c = y]];**

Select[d, StringFreeQ[#, FromCharacterCode[6]] &]]

In[2543] **:= SubsStrLim["1234363556aaa36", "3", "6"]**
Out[2543]= {"36"**,** "3556"**,** "36"}
In[2544]**:= SubsStrLim[DefOpt["SubsStrLim"], "{", "}"]**
Out[2544]= {"{}"**,** "{j+ 1, j+ 1}"**,** "{2,–2}"**,** "{y, z}"}
In[2545]**:= SubsStrLim["1234363556aaa363", "3", "3"]**
Out[2545]= {"343"**,** "363"**,** "3556aaa3"**,** "363"}

The call **SubsStrLim[*x*,*y*,*z*]** returns the list of substrings of a string***x*** that are limited by symbols {***y*, *z***} provided that these symbols don't belong to these substrings, excepting their ends. In particular, the**SubsStrLim** procedure is a quite useful means at need of extracting from of definitions of functions, blocks and modules given in string format of some components composing them that are limited by certain symbols, at times, significantly simplifying a number of procedures of processing of such definitions. Whereas, the call **SubsStrLim1[*x*, *y*, *z*]** of the procedure that is an useful modification of the previous**SubsStrLim** procedure, returns the list of substrings of a string***x*** that are limited by symbols {***y*,*z***} provided that these symbols or don't enter into substrings, excepting their ends, or along with their ends have identical number of entries of pairs {***y*, *z***} [48], for example**:**

In[2215] **:= SubsStrLim1["art[kr[xyz]sv][rans]80[[480]]", "[", "]"]** Out[2215]= {"[kr[xyz]sv]"**,** "[xyz]"**,** "[rans]"**,** "[[480]]"**,** "[480]"}
In[2216]**:= SubsStrLim1["G[x_] := Block[**{**a = 80, b = 480, c = 2014**}**,**

(a^2 + b^3 + c^4)*x]", " {", "}"]
Out[2216]= {"{a= 80**,** b= 480**,** c= 2014}"}
The mechanism of string patterns is quite often used for extraction of some structures from text strings. In a certain degree the the given mechanism we can quite consider as a

special programming language of text structures and strings. The mechanism of string patterns provides a rather serious method to make various processing of string structures. At that, acquaintance with special languages of processing of strings in many cases allows to determine string patterns by the notation of regular expressions which are determined in the**Mathematica** system on the basis of the**RegularExpression** function. The interested reader is sent to [60,71] or to the reference on the system. In this light the**RedSymbStr** procedure is represented as a quite useful means whose call**RedSymbStr[*x,y,z*]** returns result of*replacement* of all substrings consisting of a symbol***y*,** of a string***x*** onto a symbol or a string***z*.** In case of lack of occurrences of***y*** in***x*,** the procedure call returns the initial string***x*.** The fragment represents source code of the procedure with examples of use.
In[2202]**:= RedSymbStr[x_/; StringQ[x], y_ /; SymbolQ1[y], z_String] :=**

Module[{**a = StringPosition[x, y], b**}**, If[StringFreeQ[x, y], x, b = Map[#[[1]] &, a]]; b = Sort[DeleteDuplicates[Map[Length, Split[b, #2–#1 == 1 &]]], Greater]; b = Mapp[Rule, Map3[StringMultiple, y, b], z];**

In[2203] **:= RedSymbStr[“a b c d ef** Out[2203]= **“**a b c d ef gh x y z**”** In[2204]**:= RedSymbStr[“a b c d ef** Out[2204]= **“**abcdefghxyz**”**
In[2205]**:= RedSymbStr[“a b c d ef** Out[2205]= **“**aGGGbGGGcGGGdGGGefGGGghGGGxGGGyGGGz**”** In[2206]**:= RedSymbStr[“a b c d ef gh x y z”, “x”, “GGG”]** Out[2206]= **“**a b c d ef gh GGG y z**”**

So, the strings generated by earlier considered **ToString1** procedure can be called as**StringStrings***(strings of strings, or the nested strings)* as in the case of lists**;** a quite simple function can be used for their testing, whose the call **StringStringQ[*x*]** returns*True* if an expression***x*** represents a string of type *StringStrings*, and*False* otherwise. In a certain sense **ToString1** procedure generates the nested strings analogously to the nested lists, and the level of nesting of a string***x*** can be determined by the simple procedure whose call **StringLevels[*x*]** returns the nesting level of a string***x*** provided that the zero level corresponds to the standard string, i.e. a string of the form***“hhh… h”*.** The fragment below represents source codes of function**StringStringQ** and procedure**StringLevels** along with typical examples of their usage.

In[2237]**:= StringStringQ[x_] := If[! StringQ[x], False, If[SuffPref[x, “"”, 1] && SuffPref[x, “"”, 2], True, False]]**

In[2238] **:= Map[StringStringQ,** {**“"vsvartkr"”, “vsv\art\kr”, a + b, “"\"vsv\art\kr\""”**}**]**
Out[2238]= {True**,** False**,** False**,** True}

In[2703]**:= StringLevels[x_ /; StringQ[x]] := Module[**{**a = x, n =–1**}**, While[StringQ[a], a = ToExpression[a]; n++; Continue[]]; n]** In[2704]:= **Map[StringLevels,** {**“agn”, “"vsv"”, “"\"art\""”, rans**}**]** Out[2704]= {0**,** 1**,** 2**,** StringLevels[rans]}
For the purpose of*simplification* of programming of a number of procedures proved useful to define the procedure, whose call**SubsPosSymb[*x, n, y, z*]**
StringReplace[x, b]] gh x y z”, ” “, ” “]
gh x y z”, ” “, ””]
gh x y z”, ” “, “GGG”]

returns a substring of a string *x* which is limited on the right*(at the left)* by a position***n,*** and at the left*(on the right)* by a symbol from a list***y;*** in addition, search in string***x*** is done from left to right*(**z=0**)* and from right to left*(**z=1**).* The procedure call on inadmissible arguments is returned unevaluated. The following fragment represents source code of the**SubsPosSymb** procedure along with typical examples of its usage.

In[2942] **:= SubsPosSymb[x_ /; StringQ[x], n_ /; PosIntQ[n], y_ /; ListQ[y]&& DeleteDuplicates[Map[CharacterQ, y]] == {True}, z_ /; z == 0||z == 1] := Module[{a = "", k = n, b}, If[n > StringLength[x], Return[Defer[SubsPosSymb[x, n, y, z]]], While[If[z == 0, k >= 1, k <= StringLength[x]], b = StringTake[x, {k, k}]; If[! MemberQ[y, b], If[z == 0, a = b <> a, a = a <> b], Break[]]; If[z == 0, k—, k++]]; a]]**

In[2943] **:= SubsPosSymb["123456789abcdfght", 5, {"g"}, 1]**
Out[2943]= **"**56789abcdf**"**
In[2944]**:= SubsPosSymb["123456789abcdfght", 16, {"z"}, 0]**
Out[2944]= **"**123456789abcdfgh**"**

The rather simple procedure **ListStrToStr** represents undoubted interest at processing of lists in string format, more precisely, the call**ListStrToStr[*x*]** where argument***x*** has format {***"a", "b", …***} converts***x*** into string of format ***"a,b, c, …",*** if the procedure call uses only an arbitrary actual argument***x;*** if the procedure call uses an arbitrary expression as the second argument, the call returns a string of format***"abcde…".*** The following fragment represents source code of the**ListStrToStr** procedure along with examples of its usage.

In[3828]**:= ListStrToStr[x_ /; ListQ[x] && DeleteDuplicates[Map[StringQ, x]] == {True}, p___] :=**

Module[{a = ""}, If[{p}== {}, Do[a = a <> x[[k]] <> ", ", {k, Length[x]}]; StringTake[a, {1,–3}], StringJoin[x]]] In[3829]**:= ListStrToStr[{"a", "b", "c", "d", "h", "t", "k", "Art", "Kr", "Rans"}]** Out[3829]= "a, b, c, d, h, t, k, Art, Kr, Rans"
In[3830]**:= ListStrToStr[{"a*b","*","t[x]","–","(c–d)","*", "j[y]", " ==", "6"}, 6]**
Out[3830]= **"**a*b*t[x]– (c– d)*j[y]== 6**"**

The following procedure is a rather useful means for ensuring of converting of strings of a certain structure into lists of strings. In particular, such tasks arise at processing of formal arguments and local variables. This problem is solved rather effectively by the**StrToList** procedure, providing converting of strings of format**"**{*xxxxxxxx* **….** *x*}**"** into the list of strings received from a string***"xxxxxxxx* …. *x*"** parted by comma symbols**",".** In absence in an initial string of both limiting symbols {"{", "}"} the string is converted into the list of symbols according to the call**Characters[*"xxxxx* … *x*"].** The next fragment represents source code of the**StrToList** procedure with examples of its use.

In[2190] **:= StrToList[x_/; StringQ[x]] := Module[{a, b={}, c = {}, d, h, k = 1, j, y = If[StringTake[x, {1, 1}] == "{" && StringTake[x, {–1,–1}] == "}", StringTake[x, {2,–2}], x]}, a = DeleteDuplicates[Flatten[StringPosition[y, "="]] + 2]; d = StringLength[y]; If[a == {}, Map[StringTrim, StringSplit[y, ","]], While[k <= Length[a], c = ""; j = a[[k]]; For[j, j <= d, j++, c = c <> StringTake[y, {j, j}]; If[! SameQ[Quiet[ToExpression[c]], $Failed] && (j == d || StringTake[x, {j + 1, j + 1}] == ","), AppendTo[b, c–> ToString[Unique[$Art$Kr$]]]; Break[]]]; k++]; h = Map[StringTrim, StringSplit[StringReplace[y, b], ","]]; Mapp[StringReplace, h,**

RevRules[b]]]]

In[2191] **:= StrToList["Kr, a = 80, b = {x, y, z}, c = {n, m, {42, 47, 67}}"]** Out[2191]= {"Kr", "a= 80", "b= {x, y, z}", "c= {n, m, {42, 47, 67}}"} In[2192]**:= StrToList["{a, b = 80, c = {m, n}}"]**
Out[2192]= {"a", "b= 80", "c= {m, n}"}
In[2193]**:= Map[StrToList, {"{a, b, c, d}", "a, b, c, d"}]**
Out[2193]= {{"a", "b", "c", "d"}, {"a", "b", "c", "d"}}

In[2194]**:= RevRules[x_ /; RuleQ[x] || ListQ[x] && DeleteDuplicates[Map[RuleQ, x]] == {True}] := Module[{a = Flatten[{x}], b}, b = Map[#[[2]]–> #[[1]] &, a]; If[Length[b] == 1, b[[1]], b]]**

In[2195] **:= RevRules[{x–> a, y–> b, z–> c, h–> g, m–> n}]**
Out[2195]= {a–> x, b–> y, c–> z, g–> h, n–> m}
The above procedure is intended for converting of strings of format"{*x…x*}" or"*x***…**x" into the list of strings received from strings of the specified format that are parted by symbols**"="** and/or comma**",".** Fragment examples quite visually illustrate the principle of performance of the procedure along with formats of the returned results. Moreover, the fragment is ended by a quite simple and useful procedure, whose call**RevRules[*x*]** returns the rule or list of rules that are reverse to the rules determined by an argument***x*** **–** a rule of format***a*** **–>** ***b*** or their list. The**RevRules** is essentially used by the**StrToList.**

The next means are useful at work with string structures. The procedure call **StringPat[*x*, *y*]** returns the string expression formed by strings of a list***x*** and objects {**"_", "__", "___"**}**;** the call returns***x*** if***x*** **–** a string. The procedure call **StringCases1[*x*,*y*, *z*]** returns the list of the substrings in a string***x*** that match a string expression, created by the call**StringPat[*x*,*y*].** Whereas the function call**StringFreeQ1[*x*,*y*, *z*]** returns*True* if no substring in a string***x*** matches a string expression, created by the call**StringPat[*x*, *y*],** and*False* otherwise. In the fragment below, source codes of the above*three* means with examples of their usage are represented.

In[2583] **:= StringPat[x_ /; StringQ[x] || ListStringQ[x], y_ /; MemberQ[{"_", "__", "___"}, y]] := Module[{a = "", b}, If[StringQ[x], x, b =Map[ToString1, x]; ToExpression[StringJoin[Map[# <> "~~" <> y <> "~~" &, b[[1 ;;–2]]], b[[–1]]]]]]]**

In[2584]**:= StringPat[{"ab", "df", "k"}, "__"]**
Out[2584]= "ab" ~~ __ ~~ "df" ~~ __ ~~ "k"

In[2585] **:= StringCases1[x_ /; StringQ[x], y_ /; StringQ[y]||ListStringQ[y], z_ /; MemberQ[{"_", "__", "___"}, z]] := Module[{b, c = "", d, k = 1}, Sort[Flatten[Map[DeleteDuplicates, If[StringQ[y], {StringCases[x, y]}, {StringCases[x, StringPat[y, z], Overlaps–> All]}]]]]]**

In[2587]**:= StringCases1["abcdfghkaactabcfgfhkt", {"ab", "df", "k"}, "___"]**
Out[2587]= {"abcdfghk", "abcdfghkaactabcfgfhk"}

In[2588] **:= StringFreeQ1[x_ /; StringQ[x], y_ /; StringQ[y]||ListStringQ[y], z_ /; MemberQ[{"_", "__", "___"}, z]] := If[StringQ[y], StringFreeQ[x, y], If[StringCases1[x, y, z] == {}, True, False]]** In[2589]**:=**

StringFreeQ1[“abcfghkaactabcfghkt”, {“ab”, “df”, “k”}, “___”] Out[2589]= True

In[2590]**:= StringFreeQ1[“abcdfghkaactabcfghkt”, {“ab”, “df”, “k”}, “___”]** Out[2590]= False

The above means are used by a number of means of ***AVZ_Package*** package, enough frequently essentially improving the programming algorithms that deal with string expressions.

Both the system means, and our means of processing of strings represented in the present book form effective tools for processing of objects of the given type. The above means of processing of string structures similar to means of **Maple** have been based as on rather widely used standard means of system **Mathematica,** and on our means presented in the present book, very clearly demonstrating relative simplicity of programming in***Math*–**language of the means similar to means of**Maple** as its main competitor. At that, existence in**Mathematica** of rather developed set of means for operating with string patterns allows to create effective and developed systems of processing of string structures which by many important indicators surpass possibilities of**Maple.** Furthermore, the means of processing of string structures which have been programmed in the***Math***language not only are more effective at temporal relation, but also the**Mathematica** system for their programming has the advanced functional means, including rather powerful mechanism of string patterns allowing to speak about a pattern type of programming and providing developed means of processing of strings on the level which not significantly yield to specialized languages of text processing.

So, our experience of usage of both systems for programming of means for operating with string structures showed that standard means of**Maple** by many essential indicators yield to the means of the same type of the***Math*–** language. For problems of this type the***Math***language appears simpler not only in connection with more developed means, but also a procedural and functional paradigm allowing to use mechanism of pure functions. So, the present chapter represents a number of the means expanding the standard facilities of system that are oriented on work with string structures. These and other means of this type are located in our package [48]. At that, their correct use assumes that this package is uploaded into the current session.

Chapter 4. Additional means of processing of sequences and lists in the*Mathematica*software

At programming of many appendices the usage not of separate expressions, but their sets formed in the form of lists is expedient. At such organization instead of calculations of separate expressions there is an opportunity to do the demanded operations as over lists in a whole–*unified objects*– and over their separate elements. Lists of various types represent important and one of the most often used structures in**Mathematica. In Mathematica** system many functions have the*Listable*–attribute saying that an operator or block, function, module***F*** with this attribute are automatically applicable to each element of the list used respectively as their operand or argument. The call **ListableQ[*x*]** of simple function returns*True* if***x*** has***Listable*–**attribute, and *False* otherwise [48]. Meanwhile, a number of the operations having*Listable*- attribute requires compliance on length of the lists operands, otherwise the corresponding erroneous situations are initiated. With the view of removal of this shortcoming the**ListOp** procedure has been offered whose the call **ListOp[*x, y, z*]** returns the list whose elements are results of application of a procedure/function***z*** to the corresponding elements of lists***x*** and***y;*** at that, in case of various lengths of such lists the procedure is applied to both lists within their common minimum length, without causing faulty situationns. The**ListOp** procedure substantially supposes the pure functions as the***3rd*** argument what considerably allows to expand a class of functions as the***3rd*** argument. In principle,*Listable*–attribute can be ascribed to any procedure / function of arity***1,*** providing its correct call on a list as the actual argument, as the following simple example illustrates, namely**:**

In[2450] **:=** {**G[**{**a, b, c, d, h**}**], SetAttributes[G, Listable], G[**{**a, b, c, d, h**}**]**}
Out[2450]= {G[{a, b, c, d, h}], Null, {G[a], G[b], G[c], G[d], G[h]}} At the formal level for a block, function or module***F*** of arity***1*** it is possible to note the following defining relation, namely**:**

Map[*F,* {*a, b, c, d, …*}]≡ {*F*[*a*],*F*[*b*],*F*[*c*],*F*[*d*], *…*}

where in the left part the procedure***F*** can be both with*Listable* attribute, and without it whereas in the right part the existence of the*Listable* attribute for a block, module or a function***F*** is supposed. At that, for blocks, functions or modules without the*Listable* attribute for receiving its effect the system**Map** function is used. For ensuring existence of the*Listable* attribute for a block / function/module the simple**ListableC** procedure can be rather useful [48]. The**Mathematica** system at manipulation with the list structures has certain shortcomings among which impossibility of direct assignment to elements of a list of expressions is, as the following simple example illustrates**:** In[2412]**:=** {**a, b, c, d, h, g, s, x, y, z**}**[[10]] = 90**

Set **::**setps**:** {a,b,c,d,h,g,s,x,y,z} in the part assignment is not a symbol.>> Out[2412]= 90
In[2413]**:= z**
Out[2413]= z
In order to*simplify* the implementation of procedures that use similar direct assignments to the list elements, the**ListAssignP** procedure is used, whose call**ListAssignP[*x, n, y*]** returns the updated value of a list***x*** which is based on results of assignment of a value***y*** or the list of values to elements***n*** of the list***x*** where***n*** **–** one position or their list. Moreover, if the lists***n*** and***y*** have different lengths, their common minimum value is chosen. The**ListAssignP** expands functionality of the**Mathematica**, doing quite correct assignments to the list elements what the system fully doesn**'**t provide. Fragment below represents source code of the**ListAssignP** along with examples of its usage.

In[2693]**:= ListAssignP[x_ /; ListQ[x], n_ /; PosIntQ[n]||PosIntListQ[n], y_] := Module[**{**a = DeleteDuplicates[Flatten[**{**n**}**]], b = Flatten[**{**y**}**], c, k = 1**}**,**

If[a[[–1]] > Length[x], Return[Defer[ListAssignP[x, n, y]]], c = Min[Length[a], Length[b]]]; While[k <= c, Quiet[Check[ToExpression[ToString[x[[a[[k]]]]] <> " = " <> ToString1[If[ListQ[n], b[[k]], y]]], Null]]; k++]; If[NestListQ1[x], x[[–1]], x]]

In[2694] **:= Clear[x, y, z]; ListAssignP[**{**x, y, z**}**, 3, 500]**
Out[2694]= {x, y, 500}
In[2695]**:= Clear[x, y, z]; ListAssignP[**{**x, y, z**}**,** {**2, 3**}**,** {**73, 67**}**]**

Out[2695]= {x, 73, 67}
In[2696]:= **Clear[x, y, z]; ListAssignP[{x, y, z}, 3, {42, 72, 2015}]** Out[2696]= {42, 72, 2015}
Along with the**ListAssignP** procedure expediently to in addition determine simple function whose call**ListStrQ[*x*]** returns*True* if all elements of a list***x***
– expressions in string format, and*False* otherwise. The following fragment represents source code of the**ListStrQ** function with an example of its use.

In[2599] **:= ListStrQ[x_ /; ListQ[x]] := Length[Select[x, StringQ[#] &]] == Length[x] && Length[x] != 0**
In[2600]**:= Map[ListStrQ, {{"a", "b", "a", "b"}, {"a", "b", a, "a", b}, {"A", "K"}}]**
Out[2600]= {True, False, True}

The following procedure is useful enough in procedural programming, its call**ListAssign[*x*,*y*]** provides assignment of values of a list***x*** to the generated variables of***y$nnn*** format, returning the nested list, whose the first element determines list of the generated ***"y$nnn"*** variables in string format, whereas the second defines the list of the values assigned to them from the list***x*.** The **ListAssign** procedure is of interest, first of all, in problems of the dynamical generation of variables with assigning values to them. The fragment below represents source code of the procedure along with an example of its usage.

In[2221] **:= ListAssign[x_ /; ListQ[x], y_ /; SymbolQ[y]] := Module[{a ={}, b}, Do[a = Append[a, Unique[y]], {k, Length[x]}]; b = Map[ToString, a]; ToExpression[ToString[a] <> "=" <> ToString1[x]]; {b, a}]**

In[2222]**:= ListAssign[{47, 25, 18, 67, 72}, h]**
Out[2222]= {{"h$533", "h$534", "h$535", "h$536", "h$537"}, {47, 25, 18, 67, 72}}

In the **Mathematica** for grouping of expressions along with simple lists also more complex list structures in the form of the nested lists are used, whose elements are also lists*(sublists)***.** In this connection the lists of***ListList***–type, whose elements**–** sublists of*identical length* are of special interest. For simple lists the system has the testing function**;** whose call**ListQ[*x*]** returns*True***,** if ***x* –** a list, and*False* otherwise. While for testing of the*nested* lists we defined the useful enough functions**NestListQ, NestListQ1, NestQL, ListListQ** [33, 48]. These means are quite often used as a part of the testing components of headings of procedures and functions both from our***AVZ_Package*** package [48], and in various blocks, functions and modules, first of all, that are used in problems of the system character [28-33].
In addition to the above testing functions some useful functions of the same class that are quite useful in programming of means to processing of the list structures of any organization have been created [33,48]. Among them can be noted testing means such as**BinaryListQ, IntegerListQ, ListNumericQ, ListSymbolQ, PosIntQ, PosIntListQ, ListExprHeadQ.** In particular, the call **ListExprHeadQ[*v*,*h*]** returns*True* if a list***v*** contains only elements meeting the condition**Head[*a*]=*h*,** and*False* otherwise. In addition, the testing means process all elements of the analyzed list, including all its sublists of any level of nesting. The next fragment represents source code of the**ListExprHeadQ** function along with typical examples of its usage.

In[2576]**:= ListExprHeadQ[x_/; ListQ[x], h_] :=
Length[x]==Length[Select[x, Head[#]===h&]]**

In[2577] **:=** {**ListExprHeadQ[**{**a + b, c–d**}**, Plus], ListExprHeadQ[**{**a*b, c/d**}**, Times], ListExprHeadQ[**{**a^b, (c+a)^d**}**, Power]**}
Out[2577]= {True**,** True**,** True}

The above means are often used at programming of the problems oriented on processing of list structures. These and other means of the given type are located in our package [48]. In addition, their correct usage assumes that the package is uploaded into the current session.
The useful**SelectPos** function provides the choice from a list of elements by their given positions. The call**SelectPos[*x,y,z*]** returns the list with elements of a list***x*,** whose numbers of positions are different from elements of a list***y*** *(at**z=1**)* whereas at***z=2*** the list with elements of the list***x*** whose numbers of positions coincide with elements of the integer list***y*** is returned. Fragment below represents source code of the function with examples of its usage.

In[2696]**:= SelectPos[x_ /; ListQ[x], y_ /; ListQ[y] &&**

DeleteDuplicates[Map[IntegerQ[#] && # > 0 &, y]] == {**True**}**, z_ /; MemberQ[**{**1, 2**}**, z]] := Select[x, If[If[z == 2, Equal, Unequal] [Intersection[Flatten[Position[x, #]], y], **{}**], False, True] &]**

In[2697] **:= SelectPos[**{**a,b,c,d,e,f,g,h,m,n,p**}**,** {**1,3,5,7,9,11,13,15,17,19,21**}**, 2]**
Out[2697]= {a**,** c**,** e**,** g**,** m**,** p}
In[2698]**:= SelectPos[**{**a,b,c,d,e,f,g,h,m,n,p**}**,** {**1,3,5,7,9,11,13,15,17,19,21**}**, 1]**
Out[2698]= {b**,** d**,** f**,** h**,** n}
It must be kept in mind that numbers of positions of the list***y*** outside the range of positions of elements of the list***x*** are ignored, without initiating an erroneous situation what is convenient for ensuring continuous execution of appendices without processing of the situations.
For the solution of a number of the problems dealing with the nested lists, in certain cases can arise problems which aren**'**t solved by direct standard means, demanding in similar situations of programming of tasks by means which are provided by**Mathematica.** It quite visually illustrates an example of the task consisting in definition of number of elements different from the list, at each level of nesting of a list and simple list*(level of nesting**0**),* and the nested. This problem is solved by procedure whose call of**ElemLevelsN[*x*]** returns the nested list whose elements are the two–element lists whose first element determines the nesting level while the second– number of elements of this level with the type different from*List.* Procedure**ElemLevelsL** is an useful modification of the above procedure [33,48]. The following fragment represents source codes of the both procedures with examples their usage.

In[2733] **:= ElemLevelsN[x_ /; ListQ[x]] := Module[**{**a=x, c=**{}**, m=0, n, k=0**}**, While[NestListQ1[a], n = Length[Select[a, ! ListQ[#] &]]; AppendTo[c,** {**k++, n–m**}**]; m = n; a = Flatten[a, 1]; Continue[]]; Append[c,** {**k++, Length[a]–m**}**]]** In[2734]**:= L =** {**a,b,a,**{**d,c,s**}**,a,b,**{**b,c,**{**x,y,**{**v,g,z,**{**90,**{**500,**{}**,72**}}**,a,k,a**}**,z**}**,b**}**,c,b**}**;** In[2735]**:= ElemLevelsN[L]**

Out[2735] = {{0, 7}, {1, 6}, {2, 3}, {3, 6}, {4, 1}, {5, 2}, {6, 0}}
In[2736]:= **Map[ElemLevelsN, {{}, {a,b,c,d,r,t,y,c,s,f,g,h,72,90,500,s,a,q,w}}]**
Out[2736]= {{{0, 0}}, {{0, 19}}}

In[2874]:= **ElemLevelsL[x_ /; ListQ[x]] := Module[{a=x, c={}, m={}, n, k=0}, While[NestListQ1[a], n = Select[a, ! ListQ[#] &];**

AppendTo[c, {k++, MinusList[n, m]}]; m = n; a = Flatten[a, 1]; Continue[]]; Append[c, {k++, MinusList[a, m]}]] In[2875]:= **ElemLevelsL[L]**
Out[2875]= {{0, {a, b, a, a, b, c, b}}, {1, {d, s}}, {2, {x, y, z}}, {3, {v, g, k}},

{4, {90}}, {5, {500, 72}}, {6, {}}}

The following procedure provides return of all possible sublists of a nested list. The call**SubLists[*x*]** returns the list of all possible sublists of the nested list***x*,** taking into account their nesting. At that, if the list***x*** is simple, the call **SubLists[*x*]** returns the empty list, i.e. {}. The following fragment represents source code of the**SubLists** procedure with examples of its application.

In[2339]:= **SubLists[x_ /; ListQ[x]] := Module[{a, b, c = {}, k = 1},**

If[! NestListQ1[x], {}, a = ToString[x]; b = DeleteDuplicates[Flatten[StringPosition[a, "{"]]]; While[k <= Length[b], AppendTo[c, SubStrSymbolParity1[StringTake[a, {b[[k]],–1}], "{", "}"][[1]]]; k++]; DeleteDuplicates[ToExpression[c[[2 ;;–1]]]]]]

In[2340] **:= L = {a,b,a,{d,c,s},a,b,{b,c,{x,y,{v,g,z,{80,{480,{},72}},a,k,a}, },b},c,b};**
In[2341]:= **SubLists[Flatten[L]]**
Out[2341]= {}
In[2342]:= **SubLists[L]**
Out[2342]= {{d, c, s}, {b, c, {x, y, {v, g, z, {80, {480, {}, 72}}, a, k, a}}, b},

{x, y, {v, g, z, {80, {480, {}, 72}}, a, k, a}}, {v, g, z, {80, {480, {}, 72}}, a, k, a}, {80, {480, {}, 72}}, {480, {}, 72}, {}}
In[2343]:= **SubLists[{a, b, {c, d, {g, h, {g, s}}, {n, m}}, {80, 480}}]** Out[2343]= {{c, d, {g, h, {g, s}}, {n, m}}, {g, h, {g, s}}, {g, s}, {n, m}, {80, 480}}

Means of operating with levels of a nested list are of special interest. In this context the following means can be rather useful. As one of such means the **MaxLevel** procedure can be considered whose call**MaxLevel[*x*]** returns the maximum nesting level of a list***x****(in addition, the nesting level of a simple list**x** is supposed equal to zero).* At that, the**MaxNestLevel** procedure is a*equivalent* version of the previous procedure. While the call**ListLevels[*x*]** returns the list of nesting levels of a list***x*;** in addition, for a simple list or empty list the procedure call returns zero. The following fragment represents source codes of the above procedures along with typical examples of their usage.

In[2562]:= **MaxLevel[x_ /; ListQ[x]] := Module[{a = x, k = 0}, While[NestListQ1[a], k++; a = Flatten[a, 1]; Continue[]]; k]**

In[2563] **:= Map[MaxLevel, {{a,b}, {a, {b,c,d}}, {{{a,b,c}}}, {a, {{c, {d}, {{h,g}}}}}}]** Out[2563]= {0, 1, 2, 4}
In[2564]:= **MaxLevel[{a, b, c, d, f, g, h, s, r, t, w, x, y, z}]**
Out[2564]= 0

In[2581]:= **ListLevels[x_ /; ListQ[x]] := Module[{a = x, b, c = {}, k = 1}, If[! NestListQ1[x], {0}, While[NestListQ1[a], b = Flatten[a, 1]; If[Length[b] >= Length[a], AppendTo[c, k++], AppendTo[c, k]]; a = b; Continue[]]; c]]**

In[2582] **:= ListLevels[{a, b, c, d, f, g, h, s, r, t, w, x, y, z}]**
Out[2582]= {0}
In[2583]**:= ListLevels[{a,b,c, {d,f, g, {h, s, {z,y,g}, r}, t}, w, {x, {{{a,b,c}}}, y}, z}]**
Out[2583]= {1, 2, 3, 4}

In[586]**:= MaxNestLevel[L_ /; ListQ[L]] := Module[{a=Flatten[L], b=L, c=0}, While[! a == b, b = Flatten[b, 1]; c = c + 1]; c]**

In[2587] **:= L = {{a, {b, {m, {x, y, {p, q, {g, 2014}}}, n}, x}, c, {{{{{{{67, 72}}}}}}}}}; Map[MaxNestLevel, {L, {a, b, c}}]**
Out[2587]= {8, 0}

Moreover, between the above means the following defining relations take place, namely**:**
Flatten[*x*]≡Flatten[*x*, MaxLevel[*x*]] MaxLevel[*x*]≡ListLevels[*x*][[–1]]

The next rather useful procedure of work with lists has structural character, first of all, for the nested lists. Generally speaking, the call**ElemOnLevels[*x*]** returns the nested list whose elements are sublists whose first elements are levels of a nested list***x*** while the others**–** elements of these levels. For lack of elements on level***j*** the sublist has the form {***j***}**;** the call**ElemOnLevels[*x*]** on a simple list***x*** returns {***0***, ***x***}**,** i.e. the simple list has the nesting level***0*.** In the following fragment the source code of the**ElemOnLevels** procedure and typical examples of its usage are represented.

In[2736]**:= ElemOnLevels[x_List] := Module[{a, b, c, d, p = 0, k, j = 1}, If[! NestListQ1[x], Flatten[{0, x}], {a, c, d}= {x, {}, {}};**

While[NestListQ1[a], b = {p++}; For[k = 1, k <= Length[a], k++, If[! ListQ[a[[k]]], AppendTo[b, a[[k]]]; AppendTo[c, k]]]; AppendTo[d, b]; a = Flatten[Delete[a, Map[List, c]], 1]; {b, c}= {{}, {}}; j++]; AppendTo[d, Flatten[{p++, a}]]]]]

In[2737]**:= ElemOnLevels[{a, b, {c, d, {f, h, d}, s, {p, w, {n, m, r, u}, t}}, x,y,z}]**

Out[2737] = {{***0***, a, b, x, y, z}, {***1***, c, d, s}, {***2***, f, h, d, p, w, t}, {***3***, n, m, r, u}} In[2738]**:= ElemOnLevels[{a, b, c, d, f, h, d, s, p, w, n, m, r, u, t, x, y, z}]** Out[2738]= {***0***, a, b, c, d, f, h, d, s, p, w, n, m, r, u, t, x, y, z}
In[2739]**:= ElemOnLevels[{{{a, b, c, d, f, h, d, s, p, w, n, m, r, u, t, x, y, z}}}]**
Out[2739]= {{***0***}, {***1***}, {***2***, a, b, c, d, f, h, d, s, p, w, n, m, r, u, t, x, y, z}} In[2740]**:= Map[ElemOnLevels,** {{{{}}}, {}, {{{{{}}}}}}**]**
Out[2740]= {{{***0***}, {***1***}, {***2***}}, {***0***}, {{***0***}, {***1***}, {***2***}, {***3***}, {***4***}}}

For assignment of the same value to the variables can be used a very simple construction***x1* = *x2* = … = *a1*,** while for assignment to variables of different values can be used construction {***x1***,***x2***, ***x3***, **…**}={***a1***, ***a2***, ***a3***, **…**} provided that lengths of both lists are*identical***,** otherwise the erroneous situation arises. In order to eliminate this shortcoming the procedure call**ListsAssign[*x*, *y*]** can be used, returning result of assignment of values of a list***y*** to a list***x*.**

In[2766]**:= ListsAssign[x_ /; ListQ[x], y_/; ListQ[y]] := Module[{b, c, d = {}, a =**

Min[Map[Length, {x, y}]], k = 1},

If[a == 0, Return[x], Off[Set::setraw]; Off[Set::write]; Off[Set::wrsym]]; While[k <= a, {b, c}= {x[[k]], y[[k]]}; AppendTo[d, b = c]; k++]; x = {Sequences[d[[1 ;; a]]], Sequences[x[[a + 1 ;;–1]]]}; On[Set::setraw]; On[Set::write]; On[Set::wrsym]; x]

In[2767] **:= L = {x, 80, a + b, Sin, t, s}; P = {a, b, c, w, 72}; ListsAssign[L, P]**
Out[2767]= {a, 80, a+ b, Sin, 72, s}
In[2768]:= **{x, y, z, h, g, w, t}= {a, b, c}**

Set ::shape: Lists {x, y, z, h, g, w, t} and {a, b, c} are not the same shape. >> Out[2768]= {a, b, c}
In[2769]:= **ListsAssign[{x, y, z, h, g, w, t}, {a, b, c}]; {x, y, z}**
Out[2769]= {a, b, c}

In[2770]:= **ListAppValue[x_List, y_] := Quiet[x = PadLeft[{}, Length[x], y]]**
In[2771]:= **x = 80; ListAppValue[{x1, y, z, h, g, w, t}, 72]; {x1, y, z, h, g, w, t, x}**
Out[2771]= {72, 72, 72, 72, 72, 72, 72, 80}

Thus, the call **ListsAssign[*x*, *y*]** returns the list*x* updated by assignments; at that, the procedure processes the erroneous and special situations caused by the assignments***x*=*y*.** While the call**ListAppValue[*x*,*y*]** provides assignment of the same value***y*** to elements of a list***x*.** The previous fragment represents source codes of the above means along with examples of their application.

The next procedure is intended for grouping of elements of a list according to their multiplicities. The call**GroupIdentMult[*x*]** returns the nested list of the following format, namely**:**

{{{***n1***}, {***x1, x2,…, xa***}}, {{***n2***}, {***y1, y2,…, yb***}}, **…**, {{***nk***}, {***z1, z2,…, zc***}}}

where { ***xi,yj, …, zp***}**–** elements of a list*x* and {***n1,n2, …,nk***}**–** multiplicities corresponding to them {***i= 1..a, j = 1..b, …, p = 1..c***}. The following fragment represents source code of the procedure along with examples of its usage. In[2997]**:= GroupIdentMult[x_List] := Module[{a = Gather[x], b},**

b = Map[{DeleteDuplicates[#][[1]], Length[#]}&, a]; b = Map[DeleteDuplicates[#] &, Map[Flatten, Gather[b, SameQ[#1[[2]], #2[[2]]] &]]]; b = Map[{{#[[1]]}, Sort[#[[2 ;;–1]]]}&, Map[Reverse, Map[If[Length[#] > 2, Delete[Append[#, #[[2]]], 2], #] &, b]]]; b = Sort[b, #1[[1]][[1]] > #2[[1]][[1]] &]; If[Length[b] == 1, Flatten[b, 1], b]]
In[2998]:= **L = {a, c, b, a, a, c, g, d, a, d, c, a, c, c, h, h, h, h, h};**

In[2999] **:= GroupIdentMult[L]**
Out[2999]= {{{5}, {a, c, h}}, {{1}, {b, g}}, {{2}, {d}}}
In[3000]:= **GroupIdentMult[{a, a, a, a, a, a, a, a, a, a, a, a, a, a, a, a, a, a, a}]**
Out[3000]= {{19}, {a}}
In[3001]:= **GroupIdentMult[RandomInteger[42, 72]]**
Out[3001]= {{{5}, {15, 19}}, {{4}, {36}}, {{3}, {7, 9, 25, 29, 34, 38, 39, 40}},

{{2}, {1, 6, 8, 11, 12, 13, 27, 30, 35, 37}} ,
{{1}, {0, 2, 3, 4, 5, 14, 16, 17, 18, 20, 22, 24, 33, 41}}}
In[3002]:= **GroupIdentMult[{}]**

Out[3002]= {}

At that, elements of the returned nested list are sorted in decreasing order of multiplicities of groups of elements of the initial list***x.***

At processing of list structures the task of grouping of elements of the *nested* lists of*ListList*–type on the basis of***n–th*** elements of their sublists represents a quite certain interest. This problem is solved by the following procedure, whose call**ListListGroup[*x, n*]** returns the nested list– result of grouping of a***ListList***–list***x*** according to***n–th*** element of its sublists. The next fragment represents source code of the procedure along with examples of its usage.

In[2369]:= **ListListGroup[x_ /; ListListQ[x], n_ /; IntegerQ[n] && n > 0] := Module[{a = {}, b = {}, k = 1},**

If[Length[x[[1]]] < n, Return[Defer[ListListGroup[x, n]]], For[k, k <= Length[x], k++, AppendTo[b, x[[k]][[n]]]; b = DeleteDuplicates[Flatten[b]]]]; For[k=1, k <= Length[b], k++, AppendTo[a, Select[x, #[[n]] == b[[k]]&]]]; a]

In[2370] **:= ListListGroup[{{80, 2}, {480, 6}, {18, 2}, {25, 2}, {72, 6}}, 2]** Out[2370]= {{{80, 2}, {18, 2}, {25, 2}}, {{480, 6}, {72, 6}}}
In[2371]**:= ListListGroup[{{80, 2}, {480, 6}, {18, 2}, {25, 2}, {72, 67}}, 6]** Out[2371]= ListListGroup[{{80, 2}, {480, 6}, {18, 2}, {25, 2}, {72, 67}}, 6]

Whereas, on inadmissible factual arguments the procedure call is returned as an unevaluated. The given procedure is quite often used at processing of the long lists containing the repeating elements.

The following procedure expands the standard**MemberQ** function onto the nested lists, its call**MemberQ[*x,y*]** returns*True* if an expression***y*** belongs to any nesting level of a list***x,*** and*False* otherwise. Whereas the call with the third optional argument**z –** an undefinite variable**–** in addition through***z*** returns the list of***ListList***–type, the first element of each its sublist defines a level of the list***x*** whereas the second defines quantity of elements***y*** on this level provided that the main output is*True,* otherwise**z** remains undefinite. The following fragment represents source code of the procedure along with the most typical examples of its application.

In[2532]:= **MemberQL[x_ /; ListQ[x], y_, z___] := Module[{b, a = ElemOnLevels[x]}, If[! NestListQ[a], a = {a}, Null]; b = Map[If[Length[#] == 1, Null, {#[[1]], Count[#[[2 ;;–1]], y]}] &, a]; b = Select[b, ! SameQ[#, Null] && #[[2]] != 0 &]; If[b == {}, False, If[{z}!= {}&& ! HowAct[z], z = b, Null]; True]]**

In[2533] **:= MemberQL[{a, b, {c, d, {f, h, d}, s, {p, w, {n, m, r, u}, t}}, x, y, z}, d]**
Out[2533]= True
In[2534]**:= MemberQL[[{a, b, {c, d, {f,h,d}, s, {p, w, {n,m,r,u}, t}}, x, y, z}, d, z]**
Out[2534]= True
In[2535]**:= z**
Out[2535]= {{1, 1}, {2, 1}}
In[2536]**:= MemberQL[{a, b}, d, z]**
Out[2536]= False

The call **ListToString[*x, y*]** returns result of converting into an unified string of all elements of a list***x***, disregarding its nesting, that are parted by a string ***y;*** whereas a string***x*** is converted into the list of the substrings of a string***x*** parted by a string***y.*** The following fragment represents source code of the **ListToString** procedure along with typical examples of its usage.

In[2813]:= **ListToString[x_ /; ListQ[x] || StringQ[x], y_ /; StringQ[y]] := Module[{a, b = {}, c, d, k = 1},**

If[ListQ[x], a = Flatten[x]; For[k, k < Length[a], k++, c = a[[k]]; AppendTo[b, ToString1[c] <> y]]; a = StringJoin[Append[b, ToString1[a[[–1]]]]], a = FromCharacterCode[14]; d = a <> StringReplace[x, y–> a] <> a; c = Sort[DeleteDuplicates[Flatten[StringPosition[d, a]]]]; For[k = 1, k < Length[c], k++, AppendTo[b, StringTake[d, {c[[k]] + 1, c[[k + 1]]–1}]]]; ToExpression[b]]]

In[2814] **:= ListToString[{a + b, {"Agn", 67}, Kr, 18, Art, 25, "RANS", {{{Avz||72}}}}, "&"]**
Out[2814]= **"**a+ b**&**"Agn"**&**67**&**Kr**&**18**&**Art**&**25**&**"RANS"**&**Avz||72**"**
In[2815]**:= ListToString["a + b&"Agn"&67&Kr&18&Art&25&Avz || 72", "&"]**
Out[2815]= {a+ b**,** **"**Agn**"**, 67**,** Kr**,** 18**,** Art**,** 25**,** Avz|| 72}

In a number of cases exists necessity to carry out *assignments* of expressions, whose number isn't known in advance and which is defined in the course of some calculations, for example, of cyclic character, to the variables. The problem is solved by a rather simple**ParVar** procedure. The call**ParVar[*x,y*]** provides assignment of elements of a list***y*** to a list of variables generated on the basis of a symbol***x*** with return of the list of these variables in the string format. The given procedure is rather widely used in problems dealing with generating of in advance unknown number of expressions. Fragment below represents source code of the**ParVar** procedure with an example of its use.

In[2610] **:= ParVar[x_ /; SymbolQ[x], y_ /; ListQ[y]] := Module[{a={}, b, k=1}, For[k, k <= Length[y], k++, AppendTo[a, Unique[x]]]; b = ToString[a]; {b, ToExpression[b <> "=" <> ToString1[y]]}[[1]]]** In[2611]:= **W = ParVar[GS, {72, 67, 47, 25, 18}]**

Out[2611] = **"**{GS$2660**,** GS$2661**,** GS$2662**,** GS$2663**,** GS$2664}**"** In[2612]**:=** **ToExpression[W]**
Out[2612]= {72**,** 67**,** 47**,** 25**,** 18}

In a number of problems dealing with lists exists necessity of calculation of difference between***2*** lists***x*** and***y*** which is defined as a list, whose elements are included into a list***x,*** but don't belong to a list***y.*** For solution of the task the following procedure is used. The call**MinusLists[*x,y,1*]** returns result of subtraction of a list***y*** from a list***x*** that consists in deletion in the list***x*** of all occurrences of elements from the list***y.*** Whereas the call**MinusLists[*x,y,* 2]** returns result of subtraction of a list***y*** from a list***x*** which consists in parity removal from the list***x*** of entries of elements from the list***y,*** i.e. the number of the elements deleted from the list***x*** strictly correspond to their number in the list***y.*** The following fragment represents source code of the**MinusLists** procedure along with typical examples of its usage.

In[2980] **:= MinusLists[x_ /; ListQ[x], y_ /; ListQ[y], z_ /; MemberQ[{1,2}, z]]:= Module[{a, b, c, k = 1, n}, If[z == 1, Select[x, ! MemberQ[y, #] &], a = Intersection[x,**

y]; b = Map[Flatten, Map[Flatten[Position[x, #] &], a]]; c = Map[Count[y, #] &, a]; n = Length[b]; For[k, k <= n, k++, b[[k]] = b[[k]][[1 ;; c[[k]]]]]; c = Map[GenRules[#, Null] &, b]; Select[ReplacePart[x, Flatten[c]], ! SameQ[#, Null] &]]]

In[2981] **:= MinusLists[**{**a, b, c, a, d, a, b, b, a, e, c, c**}, {**a, n, b, b, b, a, d, c**}**, 2]**
Out[2981]= {a, a, e, c, c}
In[2982]**:= MinusLists[**{**a, b, c, a, d, a, b, b, a, e, c, c**}, {**a, n, b, b, b, a, d, c**}**, 1]**
Out[2982]= {e}

To the given procedure two means of **MinusList** and**MinusList1** directly adjoin which are of interest as auxiliary means at realisation of a number of our procedures [33,48], and also independent interest at processing of lists. At programming of a number of procedures of access to datafiles has been detected expediency of creation of a certain procedure useful also in other appendices. So, in this context the procedure**PosSubList** has been created, whose call**PosSubList[*x, y*]** returns a nested list of*initial* and*final* elements for entry into a simple list***x*** of a tuple of elements specified by a list***y.*** The following fragment represents source code of the**PosSubList** procedure and typical examples of its application.

In[2260]**:= PosSubList[x_ /; ListQ[x], y_ /; ListQ[y]] := Module[**{**d, a = ToString1[x], b = ToString1[y], c = FromCharacterCode[16]**}**, d = StringTake[b,** {**2,–2**}**];**

If[! StringFreeQ[a, d], b = StringReplace[a, d –> c <> "," <> StringTake[ToString1[y[[2 ;;–1]]], {**2,–2**}**]]; Map[**{**#, # + Length[y]–1**}**&, Flatten[Position[ToExpression[b], ToExpression[c]]]],** {}**]]**

In[2261] **:= PosSubList[**{**a, a, b, a, a, a, b, a, x, a, b, a, y, z, a, b, a**}, {**a, b, a**}**]**
Out[2261]= {{2, 4}, {6, 8}, {10, 12}, {15, 17}}
In[2262]**:= PosSubList[**{**a, a, b, a, a, a, b, a, x, y, z, a, b, a**}, {**a, x, a, b, c**}**]** Out[2262]= {}

The similar approach once again visually illustrates incentive motives and prerequisites for programming of the user tools expanding the**Mathematica** software. Many of means of our***AVZ_Package*** package appeared exactly in this way [28-33,48].

The procedures **Gather1** and**Gather2** a little extend the standard function **Gather1,** being rather useful in a number of appendices. The call**Gather1[*L, n*]** returns the nested list formed on the basis of the list***L*** of***ListList***–type by means of grouping of sublists of***L*** by its***n****-th* element. While call**Gather2[*L*]** returns either the simple list, or the list of***ListList***–type which defines only multiple elements of a list***L*** with their multiplicities. At absence of multiple elements in***L*** the procedure call returns the empty list, i.e. {} [48].
In[2472]**:= L =** {{**42, V, 1**}, {**47, G, 2**}, {**64, S, 1**}, {**69, V, 2**}, {**64, G, 3**}, {**44, S, 2**}}**;**

Gather1[L, 3]
Out[2472]= {{{42, V, 1}, {64, S, 1}}, {{47, G, 2}, {69, V, 2}, {44, S, 2}}, {{64, G, 3}}}
In[2473]**:= Gather2[**{**"a", 480, "a", 80, "y", 80, "d", "h", "c", "d", 80, 480**}**]**
Out[2473]= {{"a", 2}, {480, 2}, {80, 3}, {"d", 2}}
The following group of means serves for ensuring sorting of lists of various type. Among them can be noted such as**SortNL, SortNL1, SortLpos, SortLS, SortNestList.** So, the call**SortNL1[*x, p, b*]** returns result of sorting of a list***x*** of the*ListList*-type according to elements in a***p***-position of its sublists on the basis of their unique decimal codes, and***b* =**

{*Greater*|*Less*}. Whereas the call **SortNestList[*x,p,y*]** returns result of sorting of a nested*numeric* or*symbolical* list*x* by a*pth* element of its sublists according to the sorting functions***Less, Greater*** for numerical lists, and***SymbolGreater*,*SymbolLess*** for symbolical lists. In both cases a nested list with nesting level***1*** as actual argument***x*** is supposed, otherwise an initial list***x*** is returned. At that, in case of*symbolical* lists the comparison of elements is done on the basis of their codes. The next fragment represents source code of the procedure with examples of its use.

In[2738]:= **SortNestList[x_ /; NestListQ[x], p_ /; PosIntQ[p], y_] :=**

Module[{a = DeleteDuplicates[Map[Length, x]], b}, b = If[SameQ[DeleteDuplicates[Map[ListNumericQ, x]], {True}] && MemberQ[{Greater, Less}, y], y, If[SameQ[DeleteDuplicates[Map[ListSymbolQ, x]], {True}] && MemberQ[{SymbolGreater, SymbolLess}, y], y], Return[Defer[SortNestList[x, p, y]]]]; If[Min[a] <= p <= Max[a], Sort[x, b[#1[[p]], #2[[p]]] &], Defer[SortNestList[x, p, y]]]]

In[2739] **:= SortNestList[{{42, 47, 67}, {72, 67, 47}, {25, 18}}, 2, Greater]** Out[2739]= {{72, 67, 47}, {42, 47, 67}, {25, 18}}
In[2740]**:= SortNestList[{{"a", Avz, b}, {x4, Agn67, y3}, {V72, G67}, {R, Ian}},**

2, SymbolGreater]
Out[2740]= {{x4, Agn67, y3}, {"a", Avz, b}, {R, Ian}, {V72, G67}}

At that, at programming of the **SortNestList** procedure for the purpose of expansion of its applicability both onto numeric, and symbolical nested lists it was expedient to define three new functions, namely**SymbolGreater** and **SymbolLess** as analogs of the operations**Greater** and**Less** respectively, and the function whose call**ListSymbolQ[*x*]** returning*True***,** if all elements of a list***x*,** including its sublists of an arbitrary nesting level have the*Symbol*-type, otherwise the call of the**ListSymbolQ** function returns*False* [28-33,48].

The **PartialSums[*x*]** procedure of the same name with the***Maple***-procedure, similarly returns the list of the partial sums of elements of a list***x*** with one difference that at coding of symbol***x*** in*string* format the call**PartialSums[*x*]** updates the initial list***x*** in situ. The fragment represents source code of the **PartialSums** procedure along with typical examples of its usage.

In[2317]**:= PartialSums[L_ /; ListQ[L] || StringQ[L] && ListQ[ToExpression[L]]] :=**

Module[{a = {}, b = ToExpression[L], k = 1, j}, For[k, k <= Length[b], k++, AppendTo[a, Sum[b[[j]], {j, k}]]]; If[StringQ[L], ToExpression[L <> " = " <> ToString[a]], a]]

In[2318] **:= PartialSums[{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18}]**
Out[2318]= {1, 3, 6, 10, 15, 21, 28, 36, 45, 55, 66, 78, 91, 105, 120, 136, 153, 171}
In[2319]**:= GS = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18};**

PartialSums["GS"]; GS
Out[2319]= {1, 3, 6, 10, 15, 21, 28, 36, 45, 55, 66, 78, 91, 105, 120, 136, 153, 171}
In[2320]**:= SV = {a, b, c, d, e, f}; PartialSums["SV"]; SV**

Out[2320]= {a, a+b, a+b+c, a+b+c+d, a+b+c+d+e, a+b+c+d+e+f}

In a number of cases exists a need of generation of the list of variables in the form***Vk(k=1..n),*** where***V*** **–** a name and***n*** **–** a positive integer. The standard functions**CharacterRange** and**Range** of**Mathematica** don't solve the given problem therefore for these purposes it is rather successfully possible to use the procedures**Range1, Range2, Range3** and**Range3,** whose source codes along with typical examples of their usage can be found in [32,48]. The call **Range1[*J1, Jp*]** returns the list of variables in shape {***J1, J2, J3, …, Jp***}**;** at that, the actual arguments are coded in**$*xxx_yyyN*** format***(N*** **=** {***0..p***|***1..p***}) while the call**Range2[*J, p*]** returns a list of variables in standard form, providing arbitrariness in choice of identifier of a variable***J,*** namely**:** {***J1, J2, J3, …, Jp***}**;** from other party, the call**Range3[*J, p*]** returns the list in form {***J1_, J2_, J3_, …, Jp_***} where***J*** **–** an identifier and***p*** **–** an integer. At last, procedure**Range4** combines standard functions**Range** and**CharacterRange** with expansion of opportunities of the second function. More detailed description of the above procedures of*Range* type with their source codes can be found in [28-33,48]. Whereas some typical examples of their usage are given below, namely**:**

In[2420]**:= Range1[$Kr_Art1, $Kr_Art6]**
Out[2420]= {$Kr_Art1, $Kr_Art2, $Kr_Art3, $Kr_Art4, $Kr_Art5, $Kr_Art6}

In[2421] **:= Range2[Kr, 12]**
Out[2421]= {Kr1, Kr2, Kr3, Kr4, Kr5, Kr6, Kr7, Kr8, Kr9, Kr10, Kr11, Kr12} In[2422]**:= Range3[h, 12]**
Out[2422]= {h1_, h2_, h3_, h4_, h5_, h6_, h7_, h8_, h9_, h10_, h11_, h12} In[2423]**:= Range4[42, 72, 2]**
Out[2423]= {42, 44, 46, 48, 50, 52, 54, 56, 58, 60, 62, 64, 66, 68, 70, 72} The next group of facilities serves for expansion of the standard**MemberQ** function, and its means are quite useful in work with list structures. So, the **MemberQ1** procedure in a certain degree expands the standard**MemberQ** function onto the nested lists while the**MemberQ2** procedure expands the same function, taking into account number of entries of an expression into a list [33,48]. The call**MemberQ1[*L*,*x*, *y*]** returns*True* if***x*** is an element of any nesting level of a list***L****(provided that a simple list has nesting level**0**);*otherwise *False* is returned. In case of return of*True* through, the third argument***y*** the call returns the list of levels of the list***L*** that contain occurrences of***x*.** While the call**MemberQ2[*L*,*x*,*y*]** returns*True* if***x*** **–** an element of a list***L;*** otherwise *False* is returned. At that, at return of*True***,** through the***3****rd* argument***y*** the number of entries of***x*** into the list***L*** is returned. The call**MemberQ3[*x*,*y*]** returns*True* if all elements of a list***y*** belong to a list***x*** excluding nesting, and *False* otherwise. Whereas the call**MemberQ3[*x*, *y*, *t*]** with the third optional argument**–***an expression***–** returns*True***,** if the list***y*** **–** a sublist of the list***x*** at arbitrary nesting level, and*False* otherwise. At last, the call**MemberQ4[*x*,*y*]** returns*True* if at least*one* element of a list***y*** or an element***y*** belongs to a list ***x*,** and*False* otherwise. If there the***3****rd* optional argument***z*,***True* is returned only in case of number of occurrences of elements***y*** not smaller than***z*.** In addition to the above means the following***2*** simple means represent a quite certain interest. The call**MemberT[*L*,*x*]** returns total number of occurrences of an expression***x*** into a list***L*** whereas the call**MemberLN[*L*, *x*]** returns the list of*ListList*-type whose*each* sublist determines number of nesting level of the list***L*** by its first element, and number of occurrences of an expression***x*** into this nesting level by its second element.

Thus, the facilities**MemberQ1, MemberQ2, MemberQ3** and**MemberQ4** along with the means**MemberT** and**MemberLN** are useful enough in processing of lists. In principle, these means allow a number of interesting modifications significantly broadening the sphere of their application. Source codes of all mentioned means of so–called***Member***–group with examples of their use can be found in [28-33,48]. The system considers a list as the object allowing multiple occurrences into it of elements and keeping the order of elements which has been given at its definition. For determination of multiplicity of the elements entering a list it is possible to use the function**MultEntryList,** whose call**MultEntryList[*x*]** returns the*ListList*–list**;** the first element of its sublists defines an element of a list***x*,** whereas the second element determines its multiplicity in the list***x*** regardless of its nesting. Source code of the function with typical examples of its use can be found in [33,48], for example**:**

In[2720]**:= MultEntryList[{“a”,b,“a”,c,h,72,g, {“a”,b,c,g,72}, g, h, {72,g,h,72}}]**
Out[2720]= {{”a“, 3}, {b, 2}, {c, 2}, {h, 3}, {72, 4}, {g, 4}}

Unlike ***2*** standard functions**Split** and**SplitBy** the procedure call**Split1[*x*,*y*]** splits a list***x*** into sublists consisting of its elements that are located between occurrences of an element or elements of a list***y*.** If***y*** don’t belong to the list ***x*,** the initial list***x*** is returned. The following fragment represents source code of the**Split1** procedure along with typical examples of its usage.

In[2746]**:= Split1[x_List, y_] := Module[{a, b, c = {}, d, h, k = 1}, If[MemberQ3[x, y] || MemberQ[x, y], a = If[ListQ[y],**

Sort[Flatten[Map[Position[x, #] &, y]]], Flatten[Position[x, y]]]; h = a; If[a[[1]] != 1, PrependTo[a, 1]];
If[a[[–1]] != Length[x], AppendTo[a, Length[x]]]; d = Length[a]; While[k <= d–1, AppendTo[c, x[[a[[k]] ;; If[k == d–1, a[[k + 1]], a[[k + 1]]–1]]]]; k++]; If[h[[–1]] == Length[x], AppendTo[c, {x[[–1]]}]]; c, x]]

In[2747] **:= Split1[{a, a, a, b, a, b, c, d, a, b, a, b, c, d, a, b, d}, a]** Out[2747]= {{a}, {a}, {a, b}, {a, b, c, d}, {a, b}, {a, b, c, d}, {a, b, d}} In[2748]**:= Split1[{a, b, a, b, c, d, a, b, a, b, c, d, a, b, d}, {a, c, d}]** Out[2748]= {{a, b}, {a, b}, {c}, {d}, {a, b}, {a, b}, {c}, {d}, {a, b, d}, {d}} In[2749]**:= Split1[{a, b, a, b, c, d, a, b, a, b, c, d, a, b, d}, {x, y, z}]** Out[2749]= {a, b, a, b, c, d, a, b, a, b, c, d, a, b, d}

If during the work in the interactive mode diagnostic messages have a quite certain sense, in the software mode*(continuous)* of execution, for example, of procedures the messages concerning especial situations don’t have a sense, complicating software processing of such situations. In this context it is more natural to identify a special situation by return of a conveniently processed expression, for example,**$Failed.** The following procedure can serve as an example. The successful call**ElemsList[*x*, *y*]** returns the elements of a list***x*** depending on list of their positions given by a list***y*.** The list***y*** format in the general case has the view {***n1*, …, *nt*,** {***m1* ; … ; *mp***}}, returning elements of a list***x*** according to a standard relation***x*[[*n1*]] … [[*nt*]][[*m1* ;…; *mp*]].** At that, the argument***y*** allows the following formats of the coding {***n1*, …, *nt***}, {{***m1* ; … ; *mp***}}, {}; whose results of use are given in the following fragment along with source code of the**ElemsList** procedure.

In[3378]**:= ElemsList[x_/; ListQ[x], y_/; ListQ[y]] := Module[{c = ””, k = 1, a =**

```
Select[y, ! ListQ[#] &], b = Select[y, ListQ[#] &]}, If[a == {}&& b == {}, x,
If[a == {}, Quiet[Check[ToExpression[ToString[x] <> "[[" <>
StringTake[ToString[b], {3,–3}] <> "]]"], $Failed]], If[b == {}, c = ToString[x];

While[k <= Length[a], c = c <> "[[" <> ToString[a[[k]]] <> "]]"; k++];
Quiet[Check[ToExpression[c], $Failed]], c = ToString[x];
While[k <= Length[a], c = c <> "[[" <> ToString[a[[k]]] <> "]]"; k++];
Quiet[Check[ToExpression[c <> "[[" <> StringTake[ToString[b], {3,–3}] <> "]]"],
$Failed]]]]]]

In[3379] := L = {{avz, agn, vsv, art, kr}, {d,e,f,g,h, {18,25,47,52,67,72}}, {g,h,j}};
In[3380]:= ElemsList[{}, {}]
Out[3380]= {}
In[3381]:= ElemsList[L, {}]
Out[3381]= {{avz, agn, vsv, art, kr}, {d,e,f,g,h, {18, 25, 47, 52, 67, 72}}, {g,h,j}}
In[3382]:= ElemsList[L, {{1 ;; 3}}]
Out[3382]= {{avz, agn, vsv, art, kr}, {d,e,f,g,h, {18, 25, 47, 52, 67, 72}}, {g,h,j}}
In[3383]:= ElemsList[L, {2, 6, {3 ;;–1}}]
Out[3383]= {47, 52, 67, 72}
In[3384]:= ElemsList[L, {2, 6, 5}]
Out[3384]= 67
In[3385]:= ElemsList[L, {2, 80.480, 5}]
Out[3385]= $Failed

In[3386] := L[[2]][[6]][[3 ;; 0]]
Part::take: Cannot take positions 3 through 0 in {18,25,47,52,67,72}. >>
Out[3386]= {18, 25, 47, 52, 67, 72}[[3;; 0]]
In[3387]:= ElemsList[L, {2, 6, {3 ;; 0}}]
Out[3387]= $Failed
```

The following two procedures expand the system means oriented on work with list structures, giving a possibility to simplify programming*(in certain cases rather significantly)* of a number of problems that use the lists. The following fragment represents source codes of procedures along with examples of their application. The call**ReduceList[*L*,*x*,*z*,*t*]** returns the result of reducing of elements of a list***L*** that are determined by a separate element ***x*** or their list to a multiplicity determined by a separate element***z*** or their list. If elements of***x*** don't belong to the list***L***, the procedure call returns the initial list***L***. At that, if**Length[*z*] < Length[*x*]** a list***z*** is padded on the right by***1*** to the list length***x***. In addition, the*fourth* argument***t*** defines direction of reducing in the list***L****(on the left at**t = 1**and on the right at**t = 2**)*. While the call**SplitList[*L*,*x*]** returns result of*splitting* of a list***L*** onto sublists by an element or elements***x;*** at that, dividers***x*** are removed from the result. If elements***x*** don't belong to the list***L***, the procedure call returns the initial list***L***. In a number of cases both procedures are rather claimed. A number of means from our***AVZ_Package*** package rather essentially use the mentioned procedures**ReduceList** and**SplitList** [48]. These means arose in the result of programming of other our certain means.

```
In[2520]:= ReduceList[L_ /; ListQ[L], x_, z_, t_ /; MemberQ[{1, 2}, t]] :=
```

Module[{a = Map[Flatten, Map[Position[L, #] &, Flatten[{x}]]], b = {}, m = Flatten[{x}], n = Flatten[{z}], k = 1}, n = If[Length[m] > Length[n], PadRight[n, Length[m], 1], n]; For[k, k <= Length[a], k++, If[Length[a[[k]]] >= n[[k]], AppendTo[b, a[[k]]], Null]]; For[k = 1, k <= Length[a], k++, a[[k]] = If[t == 1, a[[k]] [[1 ;; Length[a[[k]]]–n[[k]]]], a[[k]][[–Length[a[[k]]] + n[[k]] ;;–1]]]]; Select[ReplacePart[L, GenRules[Flatten[a], Null]], ! SameQ[#, Null] &]]

In[2521]**:= ReduceList[**{**f, d, d, d, d, d, f, f, f, f, f, d**}, {**d, f**}**, 3, 2]** Out[2521]= {f, d, d, d}

In[2522] **:= ReduceList[**{**f, d, d, d, d, d, f, f, f, f, f, d**}, {**d, f**}**, 3, 1]** Out[2522]= {d, d, f, d}
In[2523]**:= ReduceList[**{**a, f, b, c, f, d, f, d, f, f, f, g**}, {**d, f**}, {**1, 2**}**, 1]** Out[2523]= {a, b, c, d, f, f, g}
In[2524]**:= ReduceList[**{**f, f, a, b, c, d, d, f, f, f, g, f**}, {**d, f**}, {**1, 2**}**, 2]** Out[2524]= {f, f, a, b, c, d, g}
In[2525]**:= L =** {**a, a, a, b, b, b, b, c, c, c, c, c, d, d, d, d, d, d, e, e, e, e, e, e, e**}**;**
In[2526]**:= ReduceList[L, DeleteDuplicates[L],** {**1, 2, 3, 4, 5**}**, 1]** Out[2526]= {a, b, b, c, c, c, d, d, d, d, e, e, e, e, e}

In[3340]**:= SplitList[L_/; ListQ[L], x_] := Module[**{**a = Flatten[**{**x**}**], c, d, h, b = ToString[Unique["$a"]]**}**, c = Map[ToString[#] <> b &, a]; d = StringJoin[Map[ToString[#] <> b &, L]];**

h = Select[StringSplit[d, c], # != "" &]; h = Map[StringReplace[#, b–> ","] &, h]; h = ToExpression[Map["{**" <> StringTake[#,** {**1,–2**}**] <> "**}**" &, h]]; Remove[b]; If[Length[h] == 1, h[[1]], h]]**

In[3341] **:= SplitList[**{**f, f, a, b, c, d, p, p, d, p, d, f, f, f, g, f**}, {**d, f**}**]** Out[3341]= {{a, b, c}, {p, p}, {p}, {g}}
In[3342]**:= SplitList[**{**f, f, a, b, c, d, p, d, f, f, f, g, f**}, {**h, f**}**]**
Out[3342]= {{a, b, c, d, p, d}, {g}}

A number of the additional means expanding the **Mathematica** software, in particular, for effective enough programming of problems of manipulation with list structures of various organization is presented in the given present chapter. These and other our means of the given orientation are presented a quite in details in [28-33,48]. In general,**Mathematica** provides the mass of useful and effective means of processing, except already the mentioned, of the list structures and objects that are based on structures of the given type. Being additional tools for work with lists**–***basic structures in**Mathematica*** **–** these tools are rather useful in a number of applications of various purpose. Meanwhile, other means which can be used quite successfully at processing lists of various format are also represented in the book. A number of means were already considered above, others will be considered below along with tools that are directly not associated with lists, but quite accepted for work with separate formats of lists.

Chapter 5. The additional means expanding the standard *Mathematica*functions, or its software as a whole

The string and list structures **–** some of the most important in**Mathematica** system, they both are considered in the previous*two* chapters in the context of means, additional to the system means, without regard to a large number of the standard functions of processing of structures of this type. Naturally, here it isn't possible to consider all range of system functions of this type, sending the interested reader to the help information on**Mathematica** or to the corresponding numerous literature. It is possible to find many of these editions on the web-site***http://www.wolfram.com/books.*** Having presented the means expanding the standard**Mathematica** software in the context of processing of*string* and*list* structures in the present chapter we will present the means expanding the**Mathematica** which are oriented on processing of other types of objects. First of all, we will present a number of means of bit– by–bit processing of arbitrary symbols. The**Bits** procedure quite significantly uses function**BinaryListQ,** providing a number of useful functions during of work with symbols. On the tuple of actual arguments**<*x, p*>,** where***x*** **–** a***1***–symbolical string*(character)* and***p*** **–** an integer in the range***0..8,*** the call**Bits[*x,p*]** returns binary representation of***x*** in the form of list, if***p=0,*** and***pth*** bit of such representation of a symbol ***x*** otherwise. Whereas on a tuple of the actual arguments**<*x,p*>,** where***x*** **–** a nonempty binary list of length no more than***8*** and***p = 0,*** the procedure call returns a symbol corresponding to the given binary list***x;*** in other cases the call**Bits[*x,p*]** is returned as unevaluated. The following fragment represents source code of the**Bits** procedure along with examples of its usage.

In[2819]:= **Bits[x_, P_/; IntegerQ[P]] := Module[{a, k}, If[StringQ[x] && StringLength[x] == 1, If[1 <= P <= 8,**

PadLeft[IntegerDigits[ToCharacterCode[x][[1]], 2], 8][[P]], If[P == 0, PadLeft[IntegerDigits[ToCharacterCode[x][[1]], 2], 8], Defer[Bits[x, P]]]], If[BinaryListQ[x] && 1 <= Length[Flatten[x]] <= 8, a = Length[x]; FromCharacterCode[Sum[x[[k]]*2^(a–k), {k, 1, a}]], Defer[Bits[x, P]]]]]

In[2820] **:= Map9[Bits, {"A", "A", {1,0,0,0,0,0,1}, "A", {1,1,1,1,0,1}}, {0,2,0,9,0}]**
Out[2820]= {{0, 1, 0, 0, 0, 0, 0, 1}, 1, "A", Bits["A", 9], "="}
If the previous**Bits** procedure provides rather*simple* processing of symbols, the following***2*** procedures**BitSet1** and**BitGet1** provide the expanded bit– by–bit information processing like our***Maple*** procedures. In the**Maple** we created a number of procedures***(Bit, Bit1, xbyte, xbyte1, xNB)*** that provide bit-by-bit information processing [47]**;** **Mathematica** also has similar means, in particular, the call**BitSet[*n, k*]** returns the result of setting of***1*** into a***k–th*** position of binary representation of an integer ***n.*** The following fragment represents procedure, whose call**BitSet1[*n,p*]** returns result of setting into positions of binary representation of an integer***n*** that are determined by the first elements of sublists of a nested list***p,*** {***0***|***1***}**–** values**;** at that, in case of non-nested list***p*** the value replacement only in a single position of integer***n*** is made. The**BitSet1** procedure is included in***AVZ_Package*** package [48].

In[2338]**:= BitSet1[n_ /; IntegerQ[n] && n >= 0, p_ /; ListQ[p]] :=**

Module[{b = 1, c, d, h = If[ListListQ[p], p, {p}], a = ToExpression[Characters[IntegerString[n, 2]]]}, If[ListListQ[h] && Length[Select[h, Length[#] == 2 && IntegerQ[#[[1]]] && IntegerQ[#[[2]]] && MemberQ[{0, 1}, #[[2]]] &]] == Length[h], Null, Return[Defer[BitSet1[n, p]]]]; For[b, b <= Length[h], b++, {c, d}= {h[[b]][[1]], h[[b]][[2]]}; If[c <= Length[a], a[[c]]

= d, Null]]; Sum[a[[k]]*2^(Length[a]–k), {k, Length[a]}]] In[2339]:= {**BitSet1[480, {{3,1},{6,0},{9,1}}], BitSet1[80,{4,0}], BitSet1[80,{7,1}]**} Out[2339]= {481, 80, 81}

In[2340] **:= BitSet1[480,** {{**3, 1**}, {**6, 0**}, {**9, 2**}}**]**
Out[2340]= BitSet1[480, {{3, 1}, {6, 0}, {9, 2}}]
In[89]**:= BitGet1[x___, n_ /; IntegerQ[n] && n >= 0, p_ /; IntegerQ[p] &&**

p > 0||ListQ[p]]:= Module[{**b = 1, c =** {}**, d, h = If[ListQ[p], p,** {**p**}**], a = ToExpression[Characters[IntegerString[n, 2]]]**}**, For[b, b <= Length[a], b++, c = Append[c, If[MemberQ[h, b], a[[b]], Null]]]; If[! HowAct[x], x = Length[a], Null]; Select[c, ToString[#] != "Null" &]]** In[90]:= {**BitGet1[h,80,**{**1,5,7**}**], h, BitGet1[47,** {**1,5,7**}**], BitGet1[p,480,**{**1,3,5**}**], p**} Out[90]= {{1, 0, 0}, 7, {1, 1}, {1, 1, 0}, 9}

Examples of application of the procedures**BitSet1** and**BitGet1** very visually illustrate the told. It should be noted that the**BitSet1** procedure functionally expands the standard functions**BitSet** and**BitClear** of**Mathematica** system, whereas the procedure**BitGet1** functionally extands the standard functions **BitGet** and**BitLength** of the system. The call**BitGet1[*n, p*]** returns the list of bits in the positions of binary representation of an integer***n*** that are defined by a list***p;*** in addition, in case of an integer***p*** the bit in a position***p*** of binary representation of integer***n*** is returned. While the call**BitGet1[*x,n,p*]** through a symbol***x*** in addition returns number of bits in the binary representation of an integer***n.*** Examples of the previous fragment very visually illustrate the aforesaid without any additional explanations.
In the**Mathematica** the*transformation rules* are generally determined by the **Rule** function, whose the call**Rule[*a,b*]** returns the transformation rule in the format***a –>b.*** These rules are used in transformations of expressions by the following functions***ReplaceAll, Replace, ReplaceRepeated, ReplacePart, StringReplaceList, StringCases, StringReplace*** which use either one rule, or their list as simple list, and*ListList*-list. For*dynamic* generation of such rules the**GenRules** procedure can be quite useful, whose the call**GenRules[*x, y*]** depending on a type of its arguments returns single rule or list of rules**;** the call**GenRules[*x, y, z*]** with the third optional argument***z* –** any expression– returns the list with single transformation rule or the nested list of*ListList*– type. Depending on the coding format, the procedure call returns result in the following format, namely**:**
(1)* GenRules[**{x, y, z,…***}**,*a*]**⇒ {***x –> a, y –> a, z –> a,…***}
(2)* GenRules[**{x, y, z,…***}**,*a, h*] ⇒** {{***x –> a***}, {***y –> a***}, {***z –> a***}, **…**} ***(3)* GenRules[**{***x, y, z,…***}**,** {***a, b, c,…***}**]**⇒ {***x –> a, y –> b, z –> c,…***} ***(4)* GenRules[**{***x, y, z,…***}**,** {***a, b, c,…***}**,*h*]** **⇒** {{***x –> a***}, {***y –> b***}, {***z –> c***}, **…**} ***(5)* GenRules[*x,*** {***a, b, c,…***}**]**⇒ {***x –> a***}
(6)* GenRules[*x, {***a, b, c,…***}**,*h*] ⇒** {***x –> a***}
(7)* GenRules[*x, a*]**⇒ {x –> a***}
(8)* GenRules[*x, a, h*]**⇒ {x –> a***}
The**GenRules** procedure is useful, in particular, when in some procedure it is necessary to dynamically generate the transformation rules depending on conditions.The following fragment represents source code of the**GenRules** procedure with most typical examples of its usage on all above–mentioned cases of coding of its call.

In[4040]:= **GenRules[x_, y_, z___] := Module[**{**a, b = Flatten[**{**x**}**], c = Flatten[If[ListQ /@** {**x, y**}**==** {**True, False**}**,**

PadLeft[{}, Length[x], y], {y}]]}, a = Min[Length /@ {b, c}]; b = Map9[Rule, b[[1 ;; a]], c[[1 ;; a]]]; If[{z}== {}, b, b = List /@ b; If[Length[b] == 1, Flatten[b], b]]]

In[4041]:= {**GenRules[**{**x, y, z**}, {**a, b, c**}**], GenRules[x,** {**a, b, c**}**],**

GenRules[{**x, y**}, {**a, b, c**}**], GenRules[x, a], GenRules[**{**x, y**}**, a]**} Out[4041]= {{x–>a, y–>b, z–>c}, {x–>a}, {x–>a, y–>b}, {x–>a}, {x–>a, y–>a}} In[4042]:= {**GenRules[**{**x, y, z**}, {**a, b, c**}**, 72], GenRules[x,** {**a, b, c**}**, 42],**

GenRules[x,a,6], GenRules[{**x,y**},{**a,b,c**}**,47], GenRules[**{**x,y**}**,a,67]**} Out[4042]= {{{x–> a}, {y–> b}, {z–> c}}, {x–> a}, {{x–> a}, {y–> b}}, {x–> a}, {{x–> a}, {y–> a}}}
In[4043]:= **GenRules[x_, y_, z___] := Module[**{**a, b**}**, b = If[ListQ[x] &&**

! ListQ[y], a = Map[Rule[#, y] &, x], If[ListQ[x] && ListQ[y], a = Map9[Rule, x, y], {**x–> Flatten[**{**y**}**][[1]]**}**]]; b = If[**{**z**}**!=** {}**, SplitBy[b, Head[#] & == Rule], b]; If[NestListQ[b] && Length[b] == 1, b[[1]], b]]**

In[4044] := {**GenRules[**{**x, y, z**}, {**a, b, c**}**], GenRules[x,** {**a, b, c**}**], GenRules[**{**x, y**}, {**a, b, c**}**], GenRules[x, a], GenRules[**{**x, y**}**, a]**}
Out[4044]= {{x–> a, y–> b, z–> c}, {x–> a}, Map9[Rule, {x, y}, {a, b, c}], {x–> a}, {x–> a, y–> a}}
In[4045]:= {**GenRules[**{**x, y, z**}, {**a, b, c**}**], GenRules[x,** {**a, b, c**}**], GenRules[**{**x, y, z**}, {**a, b, c**}**], GenRules[x, a], GenRules[**{**x, y**}**, a]**}
Out[4045]= {{x–> a, y–> b, z–> c}, {x–> a}, {x–> a, y–> b, z–> c}, {x–> a}, {x–> a, y–> a}}
In[4046]:= {**GenRules[**{**x, y, z**}, {**a, b, c**}**, 72], GenRules[x,** {**a, b, c**}**, 42], GenRules[x, a, 6], GenRules[**{**x, y**}**, a, 67]**}
Out[4046]= {{{x–> a}, {y–> b}, {z–> c}}, {x–> a}, {x–> a}, {{x–> a}, {y–> a}}}

In[2457]:= **GenRules2[x_ /; ListQ[x], y_] := If[ListQ[y], Map[Rule[x[[#]], y[[#]]] &, Range[1, Min[Length[x], Length[y]]]], Map[Rule[x[[#]], y] &, Range[1, Length[x]]]]** In[2458]:= {**GenRules2[**{**x, y, z**}, {**a, b, c**}**], GenRules2[**{**x, y, z**}**, h], GenRules2[**{**x, y, z**}, {**a, b**}**], GenRules2[**{**x, y, z**}, {**a, b, c, d**}**]**} Out[2458]= {{x–> a, y–> b, z–> c}, {x–> h, y–> h, z–> h}, {x–> a, y–> b}, {x–> a, y–> b, z–> c}}

The **GenRules** procedure of the same name that is functionally equivalent to the initial procedure is given as an useful modification provided that lists ***x*** and***y*** as*two* first arguments have identical length. The simple**GenRules2** function which depending on type of the*second* argument generates the list of transformation rules of the above formats***(1)*** and***(3)*** respectively finishes the fragment as illustrate very transparent examples. In certain cases these means allows quite significantly to reduce source code of the programmable procedures. Some means of our package essentially use these means [48].

Along with the considered *transformation* rules of the form***a–>b*** the system allows use also of the*delayed* rules*(RuleDelayed)* of the form***a:>b*** or***a:→b*** which are realized only at the time of their application. In the rest they are similar to the already considered transformation rules. For generation of list of transformation rules of similar type can be used the**GenRules** procedure presented above for which the***Rule*** function is replaced by the***RuleDelayed*** function, or can be used its modification**GenRules1** adapted onto usage of one or other function by the corresponding coding of the third argument at the**call GenRules1[*x*,*y*, *h*, *z*]**, where***x*, *y*, *z* –** arguments completely similar to the arguments of the same name of the procedure**GenRules** whereas the third***h*** argument determines the mode of generation of the list of the usual or delayed rules on the basis of the received value***"rd"****(delayed rule)* or***"r"*** *(simple rule)***.** The next fragment represents a source code of the**GenRules1** procedure along with the most typical examples of its usage.

In[2623]:= **GenRules1[x_, y_, h_ /; h == "r" || h == "rd", z___] := Module[{a, b = Flatten[{x}], c = Flatten[If[Map[ListQ, {x, y}] ==**

{ True, False}, PadLeft[{}, Length[x], y], {y}]]}, a = Min[Map[Length, {b, c}]]; b = Map9[If[h == "r", Rule, RuleDelayed], b[[1 ;; a]], c[[1 ;; a]]]; If[{z}== {}, b, b = Map[List, b]; If[Length[b] == 1, Flatten[b], b]]]

In[2624]:= **GenRules1[{x, y, z}, {a, b, c}, "r", 480]**
Out[2624]= {{x–> a}, {y–> b}, {z–> c}}

In[2625] := **GenRules1[{x, y, z}, {a, b, c}, "rd", 80]**
Out[2625]= {{x:→ a}, {y:→ b}, {z:→ c}}
In[2626]:= **GenRules1[{x, y, z}, a, "r"]**
Out[2626]= {x–> a, y–> a, z–> a}
In[2627]:= **GenRules1[{x, y, z}, a, "rd"]**
Out[2627]= {x:→ a, y:→ a, z:→ a}
In[2628]:= **GenRules1[{x, y}, {a, b, c, d}, "rd", 480]**
Out[2628]= {{x:→ a}, {y:→ b}}
In[2629]:= **GenRules1[x, a, "rd", 80]**
Out[2629]= {x:→ a}

Considering the importance of the ***map***function, since***Maple 10,*** the option ***`inplace`***, admissible only at usage of this function with rectangular***rtable–*** objects at renewing these objects in situ was defined. Whereas for objects of other type this mechanism isn't supported as certain examples from [25–27] illustrate. For the purpose of disposal of this shortcoming we offered a quite simple**MapInSitu** procedure [27,47]. Along with it the similar means and for**Mathematica** in the form of two functions**MapInSitu** and**MapInSitu1** together with the**MapInSitu2** procedure have been offered. The following fragment represents source codes of the above means with typical examples of their application.

In[2650]:= **MapInSitu[x_, y_/; StringQ[y]] := ToExpression[y <> "=" <> ToString[Map[x, ToExpression[y]]]]**

In[2651] := **y = {a, b, c}; h = {{4.2, 7.2}, {4.7, 6.7}};**
{**MapInSitu[G, "y"], MapInSitu[Sin, "h"]**}

Out[2651] = {{G[a], G[b], G[c]}, {{–0.871576, 0.793668}, {–0.999923, 0.40485}}}
In[2652]:= {**y, h**}
Out[2652]= {{G[a], G[b], G[c]}, {{–0.871576, 0.793668}, {–0.999923, 0.40485}}}
In[2653]:= {**H, G**}= {{**8.48, 47.67, 18.25**}, {**7.8, 47.67, 18.25**}}
Out[2653]= {0.810367,–0.519367,–0.564276}

In[2654] := **MapInSitu1[x_, y_] := ToExpression[ToString[Args[MapInSitu, 80]] <>

"=" <> ToString[Map[x, y]]]
In[2655]**:= y = {{80.42, 25.57}, {80.45, 80.89}}; MapInSitu1[Sin, y]**
Out[2655]= {{–0.95252, 0.423458}, {–0.942959,–0.711344}}
In[2656]**:= y**
Out[2656]= {–0.942959,–0.711344}
In[2657]**:= MapInSitu2[x_, y_] := Module[{a = Map[x, y], b = ToString[y], h, d = {}, k = 1, c = Select[Names["`*"], StringFreeQ[#, "$"] &]}, For[k, k <= Length[c], k++, h = c[[k]]; If[ToString[ToExpression[h]] === b, d = Append[d, h], Null]]; For[k = 1, k <= Length[d], k++, h = d[[k]]; ToExpression[h <> " = " <> ToString[a]]]; a]**
In[2658]**:= MapInSitu2[Sin, {7.4, 47.67, 18.25}]**
Out[2658]= {0.998543,–0.519367,–0.564276}
In[2659]**:= {H, G}**
Out[2659]= {{0.810367,–0.519367,–0.564276},{0.998543,–0.519367,–0.564276}} With the mechanisms used by***Maple*–**процедурой**MapInSitu** and***Math*–** functions**MapInSitu** of the same name and**MapInSitu1** can familiarize in [25-33,47,48]. Means **MapInSity** for both systems are characterized by the prerequisite, that the*second* argument at their call points out on an identifier in string format to which a certain value has been ascribed earlier and that is updated in situ after its processing by the function {***map*|Map**}. The call**MapInSitu1[*x,y*]** provides assignment to all identifiers to which in the current session the values coinciding with value of the second argument ***y*** have been ascribed, of the result of the call of**Map,** updating their values in situ. Anyway, the calls of these procedures return**Map[*x, y*]** as a result. The previous fragment represents source codes of all these procedures and typical examples of their usage.
The standard**Part** function is quite useful at analysis and processing of the expressions in addition to the**Head** function, allowing*six* formats of coding [60]. Between the functions**Head, Level** and**Part** some useful relations take place that can be used for problems of testing of expressions, in particular, **Part[*Ex, 0*]≡Head[*Ex*], Level[*Ex, 1*][[*1*]]≡Part[*Ex, 1*], Level[*Ex*,*Infinity*]≡ Level[*Ex, –1*],** where***Ex* –** an arbitrary expression, etc. The given means can be used quite successfully for testing and processing of expressions. So, the following fragment represents source code of the procedure, whose the call **Decomp[*x*]** returns the list of all unique atomic components of an arbitrary expression***x*,** including*names* of variables, functions, procedures, operations along with constants. This procedure significantly uses the above functions **Level** and**Head;** usage of the functions**Head, Level** and**Part** in a number of functions and procedures of the package [48] proved their effectiviness. In[2417]**:= Decomp[x_] := Module[{c = DeleteDuplicates[Flatten[Level[x, Infinity]], Abs[#1] === Abs[#2] &], b = {}, k}, Label[ArtKr];**
For[k = 1, k <= Length[c], k++, b = Append[b, If[AtomQ[c[[k]]], c[[k]], {Level[c[[k]],–1], Head[c[[k]]]}]]]; b = DeleteDuplicates[Flatten[b], Abs[#1] === Abs[#2] &]; If[c == b, Return[b], c = b; b = {}; Goto[ArtKr]]]

In[2418]**:= Decomp[{6*Cos[x]–n*Sin[y]/(Log[h]–b), ProcQ[c, d]}]** Out[2418]= {6, x, Cos, Times,–1, n, b, h, Log, Plus, Power, y, Sin, c, d, ProcQ}

The following procedure makes grouping of the expressions that are given by argument***L*** according to their types defined by the**Head1** procedure**;** at that, a separate expression or their list is coded as an argument***L*.** The call of **GroupNames[*L*]** returns simple list or nested list, whose elements are lists, whose*first* element– an object*type* according to the**Head1** procedure while the others– expressions of this type. The fragment represents source code of the**GroupNames** with examples from which the format of the result that is returned by the procedure is visible quite transparently.

In[2486]:= **GroupNames[L_] := Module[{a = If[ListQ[L], L, {L}], c, d, p, t, b = {{"Null", "Null"}}, k = 1},**

For[k, k <= Length[a], k++, c = a[[k]]; d = Head1[c]; t = Flatten[Select[b, #[[1]] === d &]]; If[t == {}, AppendTo[b, {d, c}], p = Flatten[Position[b, t]][[1]]; AppendTo[b[[p]], c]]]; b = b[[2 ;;–1]]; If[Length[b] == 1, Flatten[b], b]] In[2487]:= **GroupNames[{Sin, Cos, ProcQ, Locals2, 80, Map1, StrStr, 67/42, Avz, Nvalue1, a + b}]**

Out[2487] = {{System, Sin, Cos}, {Module, ProcQ, Locals2, Map1, Nvalue1}, {Integer, 80}, {Function, StrStr}, {Rational, 67/42}, {Symbol, Avz}, {Plus, a+b}} In[2488]:= **GroupNames[Head1]**
Out[2488]= {Module, Head1}

In[2489] := **L = GroupNames[Names["*"]]**
Out[2489]= {{**Global`System,** "\[FormalA]", …, "CallPacket"}, {**Function,** "AcNb", …, "$ProcName"},
{**String,** "ActionMenu", …, "GroebnerBasis"},
{**Module,** "ActiveProcess", …, "WhichN"},
{**System,** "CanberraDistance", …, "$VersionNumber"}}
In[2490]:= **Map[Length, L]–1**
Out[2490]= {594, 548, 90, 500, 6817}

In particular, from ***2*** last examples of the**GroupNames** usage follows, that names of the current session belong to five groups, namely:***Global'System, Function, String, Module*** and***System,*** the number of elements in which is ***594, 548, 90, 500*** and***6817*** respectively. Meanwhile, for receiving this result a considerable time expenditure are needed, due to the need of testing of a large number of means of the current session.
In addition to the**GroupNames** procedure a certain interest can present and a rather simple procedure, whose call**LocObj[*x*]** returns the three–element list whose*first* element defines an object***x*,** the*second* element determines its type in the context {***"Module", "SFunction"****(system function)*, ***"Expression", "Function"***}, while the*third* element– its location in the context {***"Global"***– the current session,***"System"*** – a kernel or**Mathematica** library,***"Context"*** – a system or user package that has been loaded into the current session and which contains definition of the object***x***}. The following fragment represents source code of the procedure and the most typical examples of its usage.

In[2244] := **LocObj[x_] := Module[{a = Head1[x], b},**
b[y_] := StringTake[Context[y], {1,–2}]; If[a == "Module", {x, "Module", b[x]}, If[a == "Function", {x, "Function", b[x]}, If[SystemQ[x], {x, "SFunction", b[x]},

{x, "Expression", "Global"}]]]] In[2245]:= **Map[LocObj, {PureDefinition, ProcQ, StrStr, Sin, a + b, 500}]** Out[2245]= {{PureDefinition, "Module", "AladjevProcedures"},

{ProcQ , "Module", "AladjevProcedures"}, {StrStr, "Function", "AladjevProcedures"},

{Sin, "SFunction", "System"},
{a+ b, "Expression", "Global"}, {500, "Expression", "Global"}} While the call**Names1[]** returns the nested ***returns the nested***element list whose***1st*** element defines the list of names of procedures, the***2nd*** element– the list of names of functions, the***3rd*** element– the list of names whose definitions have been evaluated in the current session whereas the***4th*** element determines the list of other names associated with the current session. The fragment represents source code of the**Names1** procedure along with an application example.

In[2545]:= **Names1[x___ /; {x}== {}] := Module[{c = 1, d, h, b = {{}, {}, {}, {}}, a = Select[Names["`*"], StringTake[#, {1, 1}] != "$" &]},**

While[c <= Length[a], d = a[[c]]; If[ProcQ[d], AppendTo[b[[1]], d], If[Quiet[Check[QFunction[d], False]], AppendTo[b[[2]], d], h = ToString[Quiet[DefFunc[d]]]; If[! SameQ[h, "Null"] && h == "Attributes[" <> d <> "] = {Temporary}", AppendTo[b[[3]], d]], AppendTo[b[[4]], d]]]; c++]; b] In[2546]:= **Names1[]**

Out[2546]= {{"Bt", "Mas", "Names1", "W"}, {"F", "G"}, {"Art25$", "Kr18"}, {}}

The **Names1** procedure is a rather useful means in a number of appendices, in particular, in some questions of the procedural programming, in certain relations expanding the standard**Names** function of**Mathematica.** Though, during work in the current session, the execution of the**Names1** procedure demands the increasing time expenditure, assuming its circumspect usage.

The call **RemoveNames[]** provides removal from the current session of the names, whose types are other than procedures and functions, and whose definitions have been evaluated in the current session; moreover, the names are removed so that aren't recognized by**Mathematica** any more. The call **RemoveNames[]** along with removal of the above names from the current session returns the nested***session returns the nested***element list whose first element determines the list of names of procedures, whereas the second element– the list of names of functions whose definitions have been evaluated in the current session. The following fragment represents source code of the**RemoveNames** with typical examples of its usage.

In[2565]:= **RemoveNames[x___] := Module[{a = Select[Names["`*"], ToString[Definition[#]] != "Null" &], b}, ToExpression["Remove[" <> StringTake[ToString[MinusList[a, Select[a, ProcQ[#]||! SameQ[ToString[Quiet[DefFunc[#]]], "Null"]|| Quiet[Check[QFunction[#], False]] &]]], {2,–2}] <> "]"]; b = Select[a, ProcQ[#] &]; {b, MinusList[a, b]}]**

In[2566] := {**Length[Names["`*"]], RemoveNames[], Names["`*"]**} Out[2566]= {80, {{"Ar", "Kr", "Rans"}, {"Ian"}}, {"Ar", "Kr", "Rans", "Ian"}} In[2567]:= **RemoveNames[]**
Out[2567]= {{"Art", "Kr", "Rans"}, {"Ian"}}
In[2568]:= **RemoveNames[]**
Out[2568]= {{"M", "M1", "M2"}, {"F", "F42", "F47", "$LoadContexts"}}

The **RemoveNames** procedure is a rather useful means in some appendices connected with cleaning of the working**Mathematica** area from definitions of non–used symbols.

The given procedure confirmed a certain efficiency in management of random access memory.
Using our procedures and functions such as**DefFunc3, HeadPF, ToString1, SymbolQ** and**PrefixQ,** it is possible to obtain the more developed means of testing of program objects of the**Mathematica;** the**ObjType** procedure acts as a similar means. The call**ObjType[*x*]** returns the type of an object***x*** in the context {*Function*,*Module*,*Block or DynamicModule*}, in other cases the type of an expression assigned in the current session to a symbol***x*** by assignment operators {:=, =} is returned. The following fragment represents source code of the**ObjType** procedure along with typical application examples.

In[2220]:= **ObjType[x_] := Module[{a, b, c, d = {}, h}, If[ToString1[HeadPF[x]] === "HeadPF[" <> ToString1[x] <> "]" ||**

SymbolQ[HeadPF[x]], Return[Head[x]], b = {ToString1[DefFunc[x]]}; c = Length[b]]; Do[AppendTo[d, h = StringSplit[b[[k]], " := "]; {h[[1]], If[PrefixQ["Module[{", h[[2]]], Module,

If[PrefixQ["Block[{", h[[2]]], Block, If[PrefixQ["Function[", h[[2]]], Function, If[PrefixQ["DynamicModule[{", h[[2]]], DynamicModule, {Function, Head[ToExpression[h[[2]]]]}]]]]}]; Flatten[d, 1]]

In[2221] **:= Sv[x_, y_] := x + y; G[x_] := Block[{}, x^2]; V[x_] := If[EvenQ[x], x, 2*x]; V[x_, y_] := Block[{a = If[PrimeQ[x], NextPrime[y]]}, a*(x + y)];**
In[2222]:= **Map[ObjType, {ObjType, 80, a + b, ProcQ}]**

Out[2222] = {{"ObjType[x_]", Module}, Integer, Plus, {"ProcQ[x_]", Module}}
In[2223]:= **Map[ObjType, {Sv, G, V}]**
Out[2223]= {{"Sv[x_, y_]", {Function, Plus}}, {"G[x_]", Block},

{ "V[x_, y_]", Block}, {"V[x_]", {Function, Times}}} In[2224]:= **ObjType[DefFunc3]**
Out[2224]= {"DefFunc3[x_ /; BlockFuncModQ[x]]", Module}
In[2225]:= **F := Function[{x, y}, x + y]; {F[80, 480], ObjType[F]}** Out[2225]= { 560, Function}
In[2226]:= **F1 := #1 * #2 &; {F1[80, 480], ObjType[F1]}**
Out[2226]= {38400, Function}
In[2227]:= **Map[ObjType, {HeadPF, StrStr}]**
Out[2227]= {{"Head1[x_]", Module}, {"StrStr[x_]", {Function, String}}} In[2228]:= **Agn := "4247679886"; Avz = 2014; Map[ObjType, {Agn, Avz}]** Out[2228]= {String, Integer}

Here is quite appropriate to make one explanation : the**ObjType** procedure carries to the***Function*** type not only especially functional objects, but also definitions of the format**Name[*x_*,*y_*, *z_*, …]: =*Expression;*** in this case the call returns the list of the following format, namely: **{"Name[*x_*,*y_*,*z_*,…]", {*Function*,*Head*[*Expression*]}}**. Due to the aforesaid the**ObjType** procedure is represented to us as a rather useful means at testing of objects of various type in the current session in problems of procedural programming.

In a number of cases exists an urgent need of determination of the program objects and their types activated directly in the current session. The problem is solved by the**TypeActObj** procedure, whose call**TypeActObj[]** returns the nested list, whose sublists in string format by the first element contain types of active objects of the current session, whereas other elements of the sublist are names corresponding to this type**;** at that, the types recognized by the system, or the types of expressions determined by us, in particular, {*`Procedure`,`Function`*} can act as a type. In a certain sense the**TypeActObj** procedure supplements the **ObjType** procedure. The following fragment represent source code of the procedure with examples of its application.

In[2787] **:= TypeActObj[] := Module[{a = Names["`*"], b ={}, c, d, h, p, k =1}, Quiet[For[k, k <= Length[a], k++, h = a[[k]]; c = ToExpression[h]; p = StringJoin["0", ToString[Head[c]]]; If[! StringFreeQ[h, "$"] || (p === Symbol && "Definition"[c] === Null), Continue[], b = Append[b, {h, If[ProcQ[c], "0Procedure", If[Head1[c] === Function, "0Function", p]]}]]]]; a = Quiet[Gather1[Select[b, ! #1[[2]] === Symbol &], 2]]; a = ToExpression[StringReplace[ToString1[DeleteDuplicates /@ Sort /@ Flatten /@ a], "AladjevProcedures`TypeActObj`"–> ""]]; Append[{}, Do[a[[k]][[1]] = StringTake[a[[k]][[1]], {2,–1}], {k, Length[a]}]]; a]**

In[2788] **:= TypeActObj[]**
Out[2788]= {{"Symbol**"**, **"**A**"**, **"**B**"**, **"**g**"**, **"**H3**"**, **"**m**"**, **"**n**"**, **"**PacletFind**"**, **"**System**"**, **"**Procedure"}, {"Procedure**"**, **"**As**"**, **"**Kr"}, {"Function**"**, **"**G**"**, **"**V"}, {"List**"**, **"**xyz"}}
In[2789]**:= TypeActObj[]**
Out[2789]= {{"String**"**, **"**Agn"}, {"Symbol**"**, **"**atr**"**, **"**F2**"**, **"**F47**"**, **"**M**"**, **"**Sv**"**, **"**V"},

{"Integer**"**, **"**Avz"}, {"Function**"**, **"**F**"**, **"**F1"},
{"Procedure**"**, **"**G**"**, **"**M2**"**, **"**M3**"**, **"**M4**"**, **"**M5**"**, **"**RemoveNames"}}

In the context of use of the standard functions **Nest** and**Map** for definition of new pure functions on the basis of available ones, it is possible to offer a procedure as an useful generalization of the standard**Map** function, whose call**Mapp[*F, E, x*]** returns result of application of a function/procedure***F*** to an expression***E*** with transfer to it of the actual arguments determined by a tuple of expressions***x*** which can be and empty. In case of the empty tuple***x*** the identity**Map[*F, E*]≡Mapp[*F, E*]** takes place. As formal arguments of the standard function**Map[*f,g*]** act the name***f*** of a procedure/function whereas as the*second* argument**–** an arbitrary expression***g*,** to whose operands of the first level is applied***f*.** The following fragment represents source code of the **Mapp** procedure along with typical examples of its usage.

In[2634] **:= Mapp[f_ /; ProcQ[f]||SysFuncQ[f]||SymbolQ[f], Ex_, x___] := Module[{a = Level[Ex, 1], b = {x}, c = {}, h, g = Head[Ex], k = 1}, If[b == {}, Map[f, Ex], h = Length[a]; For[k, k <= h, k++, AppendTo[c, ToString[f] <> "[" <> ToString1[a[[k]]] <> ", " <> ListStrToStr[Map[ToString1, {x}]] <> "]"]]; g[Sequences[ToExpression[c]]]]]** In[2635]**:= Mapp[F, {a, b, c}, x, y, z]**
Out[2635]= {F[a**,** x**,** y**,** z], F[b**,** x**,** y**,** z], F[c**,** x**,** y**,** z]}

In[2636] **:= Mapp[F, a + b + c, x, y, z]**
Out[2636]= F[a**,** x**,** y**,** z]+ F[b**,** x**,** y**,** z]+ F[c**,** x**,** y**,** z]
In[2637]**:= Mapp[F, (m + n)/(g + h) + Sin[x], a, b, c]**
Out[2637]= F[(m+ n)/(g+ h)**,** a**,** b**,** c]+ F[Sin[x]**,** a**,** b**,** c]
In[2638]**:= Mapp[StringPosition, {"11123", "33234"}, {"2", "3", "23"}]** Out[2638]=

{{{4, 4}, {4, 5}, {5, 5}}, {{1, 1}, {2, 2}, {3, 3}, {3, 4}, {4, 4}}} In[2639]:= **Mapp[StringReplace, {"123525", "2595"}, {"2"–> "V", "5"–> "G"}]** Out[2639]= {"1V3GVG", "VG9G"}
In[2640]:= **Map[F, {{a, b}, {c, d, e}}]**
Out[2640]= {F[{a, b}], F[{c, d, e}]}
In[2641]:= **Mapp[F, {{a, b}, {c, d, e}}, x, y, z]**
Out[2641]= {F[{a, b}, x, y, z], F[{c, d, e}, x, y, z]}
In[2642]:= **Mapp[ProcQ, {Sin, ProcQ, Mapp, PureDefinition, SysFuncQ}]**
Out[2642]= {False, True, True, True, False}

In[2653]:= **Mapp1[f_ /; SymbolQ[f], L_ /; ListQ[L]] := Module[{b, a = Attributes[f]}, SetAttributes[f, Listable]; b = Map[f, L]; ClearAllAttributes[f]; SetAttributes[f, a]; b]**

In[2654] **:= Map[F, {{a, b, c}, {x, y, {c, d, {h, k, t}}}}]**
Out[2654]= {F[{a, b, c}], F[{x, y, {c, d, {h, k, t}}}]}
In[2655]:= **Mapp[F, {{a, b, c}, {x, y, {c, d, {h, k, t}}}}]**
Out[2655]= {F[{a, b, c}], F[{x, y, {c, d, {h, k, t}}}]}
In[2656]:= **Mapp1[F, {{a, b, c}, {x, y, {c, d, {h, k, t}}}}]**
Out[2656]= {{F[a], F[b], F[c]}, {F[x], F[y], {F[c], F[d], {F[h], F[k], F[t]}}}}

We will note that realization of algorithm of the**Mapp** procedure is based on the following relation, namely:
Map[*F, Expr*]≡Head[*Expr*][Sequences[Map[*F*,Level[*Expr*,*1*]]]]

Whose *rightness* follows from definition of the system functions**Head, Map, Level,** and also of the**Sequences** procedure considered in the present book. The following simple example rather visually illustrates the aforesaid:

In[4942] **:= Map[F, (m + n)/(g + h) + Sin[x]] == Head[(m + n)/(g + h) + Sin[x]][Sequences[Map[F, Level[(m + n)/(g + h) + Sin[x], 1]]]]**
Out[4942]= True
The given relation can be used and at realization of cyclic structures for the solution of problems of other directionality, including programming on the basis of use of mechanism of the*pure* functions. While the**Mapp** procedure in some cases rather significantly simplifies programming of various tasks. The***Listable*** attribute for a function***F*** determines that the function***F*** will be automatically applied to elements of the list that acts as its argument. Such approach can be used rather successfully in a number of cases of realization of blocks, functions and modules.

In particular, in this context a rather simple **Mapp1** procedure is of interest, whose call**Mapp1[*x, y*]** unlike the call**Map[*x, y*]** of the standard function returns result of applying of a block, function or a module***x*** to all elements of a list***y,*** regardless of their location on list levels. The previous fragment represents source code of the**Mapp1** procedure with comparative examples relative to the system**Map** function.

Meanwhile, for a number of functions and expressions the *Listable*-attribute doesn't work, and in this case the system provides***2*** special functions**Map** and**Thread** that in a certain relation can quite be referred to the structural means that provide application of functions to parts of expressions. In this conexion we created group of enough simple and at the same time useful procedures and functions, so-called*Map*means which enough significantly expand the system**Map** function. Two means of this group– the procedures **Mapp** and**Mapp1** that have a number of applications in means of package ***AVZ_Package*** [48] have already been presented above, we will present also other means of this***Map***–group. The following fragment represents source codes of means of this group with typical examples of their application that on the formal level rather visually illustrate results of calls of these means on correct factual arguments. Similar representation allows to significantly minimize descriptions of means when on the basis of formal results of calls it is quite simple to understand an essence of each means of***Map***–group.

In[2625]:= **Map1[x_ /; ListQ[x] && SameQ[DeleteDuplicates[Map[SymbolQ[#] &, x]], {True}], y_List] := Map[Symbol[ToString[#]][Sequences[y]] &, x]** In[2626]:= **Map1[{F, G, H, V}, {x, y, z, h, p, t}]**
Out[2626]= {**F**[x,y,z,h,p, t], **G**[x,y,z,h,p,t], **H**[x,y,z,h,p,t], **V**[x,y,z,h,p,t]}

In[2627] := **Map2[F_ /; SymbolQ[F], c_ /; ListQ[c], d_ /; ListQ[d]] := Map[Symbol[ToString[F]][#, Sequences[d]] &, c]**
In[2628]:= **Map2[F, {a, b, c, d, e, g}, {x, y, z, p, q, h}]**
Out[2628]= {**F**[**a,** x, y, z, p, q, h], **F**[**b,** x, y, z, p, q, h], **F**[**c,** x, y, z, p, q, h], **F**[**d,** x, y, z, p, q, h], **F**[**e,** x, y, z, p, q, h], **F**[**g,** x, y, z, p, q, h]}
In[2629]:= **Map3[f_ /; SymbolQ[f], g_, l_ /; ListQ[l]] := Map[Symbol[ToString[f]][g, #] &, l]**
In[2630]:= **Map3[F, H, {x, y, z, h, p, h, m, n}]**
Out[2630]= {**F**[**H**,x], **F**[**H**,y], **F**[**H**,z], **F**[**H**,h], **F**[**H**,p], **F**[**H**,h], **F**[**H**,m], **F**[**H**,n]}

In[2631] := **Map4[F_ /; SymbolQ[F], L_/; ListQ[L], x_] := Map[Symbol[ToString[F]][#, x] &, L]**
In[2632]:= **Map4[F, {a, b, c, d, h, g, m, n}, x]**
Out[2632]= {F[a, x], F[b, x], F[c, x], F[d, x], F[h, x], F[g, x], F[m, x], F[n, x]}

In[2633]:= **Map5[F_, L_ /; NestListQ[L]] := Map[F[Sequences[#]] &, L]** In[2634]:= **Map5[S, {{x1, y1, z1, t1}, {x2, y2, z2}, {x3, y3}, {x4, y4, z4, t4, m, n}}]**

Out[2634] = {S[x1, y1, z1, t1], S[x2, y2, z2], S[x3, y3], S[x4, y4, z4, t4, m, n]}
In[2635]:= **F[x_, y_, z_, h_] := a[x]*b[y]*d[z]*g[z]–c[x, y, z]**
In[2636]:= **Map5[F, {{x1, y1, z1, t1}, {x2, y2, z2}, {x3, y3}, {x4, y4, z4, t4, m, n}}]**
Out[2636]= {–c[x1, y1, z1]+ a[x1] b[y1] d[z1] g[z1], F[x2, y2, z2], F[x3, y3],

F[x4, y4, z4, t4, m, n]}

In[2637] := **Map6[F_ /; PureFuncQ[F], L_ /; ListListQ[L]] := Module[{a, h, p, b = Length[L], c = Length[L[[1]]], d = {}, k = 1}, h = StringTake[ToString[F], {1,–4}]; For[k, k <= b, k++, a = {}; AppendTo[d, StringReplace[h, Flatten[{For[p = 1, p <= c, p++, AppendTo[a, "#" <> ToString[p]–> ToString[L[[k]][[p]]]]], a}][[2 ;;–1]]]]]; ToExpression[d]]**

In[2638] := **Map6[a[#1]*b[#2]*d[#3]*g[#4z]–c[#1, #2, #3] &, {{x1, y1, z1, t1}, {x2, y2, z2, t2}, {x3, y3, z3, t3}, {x4, y4, z4, t4}}]**
Out[2638]= {–c[x1, y1, z1]+ a[x1] b[y1] d[z1] g[t1],–c[x2, y2, z2]+a[x2] b[y2] d[z2] g[t2],–c[x3, y3, z3]+ a[x3] b[y3] d[z3] g[t3]–c[x4, y4, z4]+ a[x4] b[y4] d[z4] g[t4]}

```
In[ 2639]:= Map7[x__ /; DeleteDuplicates[Map[SymbolQ, {x}]] === {True}, y_ /;
ListQ[y]] := Map[FunCompose[Reverse[Map[Symbol, Map[ToString, {x}]]], #] &, y]
In[2640]:= Map7[F, G, H, {a, b, c, d, h}]

Out[ 2640]= {F[G[H[a]]], F[G[H[b]]], F[G[H[c]]], F[G[H[d]]], H[G[F[h]]]} In[2641]:=
Map7[Sin, Sqrt, N, {18, 25, 47, 67, 72, 480}]
Out[2641]= {–0.891682,–0.958924, 0.541709, 0.945597, 0.807261, 0.0821536}

In[2642]:= Map8[x__ /; DeleteDuplicates[Map[SymbolQ, {x}]] === {True}, y_ /;
ListQ[y]] := Map[Symbol[ToString[#]][Sequences[y]] &, {x}] In[2643]:= Map8[x, y,
z, h, g, {a, b, c, d}]
Out[2643]= {x[a, b, c, d], y[a, b, c, d], z[a, b, c, d], h[a, b, c, d], g[a, b, c, d]} In[2644]:=
Map9[F_ /; SymbolQ[F], x_ /; ListQ[x], y_ /; ListQ[y]] := If[Length[x] == Length[y],
Map13[F, {x, y}], Defer[Map9[F, x, y]]]

In[2645] := Map9[F, {a, b, c, d, g, p}, {x, y, z, h, s, w}]
Out[2645]= {F[a, x], F[b, y], F[c, z], F[d, h], F[g, s], F[p, w]}
In[2646]:= Map9[Rule, {“72a”, “67g”, “47s”, “80b”}, {“a”, “b”, “c”, “d”}]
Out[2646]= {”72a”–> “a“, “67g”–> “b“, “47s”–> “c“, “80b”–> “d”} In[2647]:=
Map9[Rule, {a, b, c, d, m, p}, {x, y, z, t, n, q}]
Out[2647]= {a–> x, b–> y, c–> z, d–> t, m–> n, p–> q}
In[2648]:= Map9[Plus, {a, b, c, d, g, p, u}, {x, y, z, h, s, w, t}]
Out[2648]= {a+ x, b+ y, c+ z, d+ h, g+ s, p+ w, u+ t}

In[2649]:= Map10[F_ /; SymbolQ[F], x_, L_ /; ListQ[L], y___] :=
Map[Symbol[ToString[F]][x, #, Sequences[{y}]] &, L]

In[2650] := Map10[F, x, {a, “b”, c, d}, y, “z”, h]
Out[2650]= {F[x,a,y, “z“,h], F[x, “b“, y, “z“, h], F[x, c, y, “z“,h], F[x,d, y, “z“,h]}
In[2651]:= Map10[F, “x”, {a, “b”, c, d, f, g}]
Out[2651]= {F[”x“, a], F[”x“, “b”], F[”x“, c], F[”x“, d], F[”x“, f], F[”x“, g]} In[2652]:=
Map10[SuffPref, “C:\89b8fc17cbdce3\ mxdwdrv.dll”, {“.nb”,

“.m”, “.dll”, “.cdf”}, 2]
Out[2652]= {False, False, True, False}

In[2653] := Map11[x_/; SymbolQ[x], y_/; ListQ[y], z_] := (If[ListQ[#1], (x[#1, z] &)
/@ #1, x[#1, z]] &) /@ y
In[2654]:= Map11[G, {x, y, z, m, n, g}, t]
Out[2654]= {G[x, t], G[y, t], G[z, t], G[m, t], G[n, t], G[g, t]}

In[2655]:= Map12[F_ /; SymbolQ[F], x_ /; NestListQ1[x]] := Module[{c, a =
ToString1[x], b = ToString[F] <> “@”}, c = StringReplace[a, {”{”–> “{” <> b, “, “–>
“,” <> b}]; c = StringReplace[c, b <> “{”–> “{“]; ToExpression[c]] In[2656]:=
Map12[F, {{a, b, c}, {x, y, z}, h, {m, {{“p”}}, n, p, {{{x, “y”}}}}}] Out[2656]=
{{F[a], F[b], F[c]}, {F[x], F[y], F[z]}, F[h], {F[m], {{F[”p”]}}, F[n],

F[p] , {{{F[x], F[y]}}}}}
In[2657]:= Map12[ToString1, {{a, b, c}, {x, y, z}, “h”, {m, {“x”}, n, p}}] Out[2657]=
{{”a“, “b“, “c”}, {”x“, “y“, “z”}, “"h"”, {”m“, {“"x"”}, “n“, “p”}}

In[2658]:= Map13[x_ /; SymbolQ[x], y_ /; ListListQ[y]] := Module[{k, j, a =
```

Length[y], b = Length[y[[1]]], c = {}, d = {}},

For[k = 1, k <= b, k++, For[j = 1, j <= a, j++, AppendTo[c, y[[j]][[k]]]]; AppendTo[d, Apply[x, c]]; c = {}]; d] In[2659]:= **Map13[F, {{a, b, c, s}, {x, y, z, g}, {m, n, p, w}}]**
Out[2659]= {F[a, x, m], F[b, y, n], F[c, z, p], F[s, g, w]}
In[2660]:= **Map13[ProcQ, {{ProcQ}}]**
Out[2660]= {True}
In[2661]:= **Map13[Plus, {{a, b, c, g, t}, {x, y, z, g, t}, {m, n, p, h, g}}]** Out[2661]= {a+ m+ x, b+ n+ y, c+ p+ z, 2 g+ h, g+ 2 t}
In[2662]:= **G[x_, y_] := x + y; Map13[G, {{a, b, c}, {x, y, z}, {m, n, p}}]** Out[2662]= {G[a, x, m], G[b, y, n], G[c, z, p]}
In[2663]:= **Map13[G, {{a, b, c, g, h}, {x, y, z, t, v}}]**
Out[2663]= {a+ x, b+ y, c+ z, g+ t, h+ v}
In[2664]:= **Map14[x_ /; SymbolQ[x], y_ /; ListQ[y], z_, t___] :=**

Module[{a = Map[x[#, z] &, y]}, If[{t}== {}, a, Map[ToString, a]]]

In[2665] := **Map14[G, {a, b, c, d, f, g, h}, Kr]**
Out[2665]= {G[a, Kr], G[b, Kr], G[c, Kr], G[d, Kr], G[f, Kr], G[g, Kr], G[h, Kr]}
In[2666]:= **Map14[G, {a, b, c, d, f, g}, Kr, 500]**
Out[2666]= {"G[a,Kr]", "G[b,Kr]", "G[c,Kr]", "G[d,Kr]", "G[f,Kr]", "G[g,Kr]"}

In[2667]:= **Map14[G, {}, Kr, 90]**
Out[2667]= {}
In[2668]:= **Map15[x__ /; SameQ[DeleteDuplicates[Map[SymbolQ, {x}]], {True}], y_] := Map[#[y] &, {x}]**

In[2669] := **Map15[TableForm, MatrixForm, {{1, V, 72}, {2, G, 67}, {3, S, 47}, {4, A, 25}, {5, K, 18}}]** **1 V 72 1 V 72**
2 G 67 2 G 67
3 S 47 , 3 S 47
4 A 25 4 A 25

Out[2669] = **5 K 18 5 K 18**
In[2670]:= **Map15[F, G, H, P, Q, X, Y, (a + b)]**
Out[2670]= {F[a+ b], G[a+ b], H[a+ b], P[a+ b], Q[a+ b], X[a+ b], Y[a+ b]}

In[2671]:= **Map16[f_/; SymbolQ[f], l_/; ListQ[l], x___] := Quiet[(f[#1, FromCharacterCode[6]] &) /@ l /. FromCharacterCode[6]–> Sequence[x]]**
In[2672]:= **Map16[F, {x, y, z, t}, h, m, p]**
Out[2672]= {F[x, h, m, p], F[y, h, m, p], F[z, h, m, p], F[t, h, m, p]} In[2673]:= **Map17[x_, y_ /; RuleQ[y] || ListRulesQ[y]] :=**
If[RuleQ[y], Map[x, y], Map[Map[x, #] &, y]] In[2674]:= **Map17[F, {a–> b, c–> d, t–> g, w–> v, h–> 80}]**
Out[2674]= {F[a]–>F[b], F[c]–>F[d], F[t]–>F[g], F[w]–>F[v], F[h]–>F[80]}

The previous fragment represents source codes of means of the above ***Map–*** group with examples of their usages, from which the structure of the results returned by them is quite visually visible. Without increasing essence, we will give only short explanations concerning means of the given group. For example, the following calls

Map1[{*F, G, H, …*}, {*x, y,…*}], Map2[*F,* {*a, b,…*}, {*x, y,…*}], Map3[*F, H,* {*x, y,…*}]

return respectively lists of the following format, namely**:**

{***F*[*x, y,…*],*G*[*x, y,…*],*H*[*x, y,…*], …**}; {***F*[*a, x, y…*],*F*[*b, x, y,…*],*F*[*c, x, y,…*], …**}; {***F*[*H, x*],*F*[*H, y*],*F*[*H, z*],*F*[*H, h*], …,*F*[*H, g*],*F*[*H, m*],*F*[*H, n*], …**}**.**

The call **Map4[*x, y, z*]** returns result in the format {***x*[*a1,z*],*x*[*a2,z*],*x*[*a3,z*], …**}**,** where***y*** = {***a1, a2, a3, …***}. Whereas two procedures**Map5** and**Map6** expand action of the system function**Map** onto cases of classical and pure functions with any number of arguments. The call**Map7[*F, G, …, V,*{*a, b, …, v*}]** where ***F,G,…,V*** **–** symbols and {***a,b,c,…,v***}**–** the list of arbitrary expressions, returns result of the following format, namely**:**

{ ***F*[*G*[…*V*[*a*]]]]…],*F*[*G*[…*V*[*b*]]]] …],*F*[*G*[…*V*[*c*]]]] …],…,*F*[*G*[…*V*[*v*]]]]…]**} without demanding any additional explanations in view of transparency.

Quite certain interest is represented by quite simple **Map8** function, whose call**Map8[*F, G, H, …, V,*** {***a, b, …, v***}**],** where to***F, G, …, V*** **–** symbols whereas {***a, b, c, …, v***}**–** the list of arbitrary expressions, returns result of the format**:**

{***F*[*a, b, c, …, v*],*G*[*a, b, c, …, v*],*H*[*a, b, c, …, v*], …,*V*[*a, b, c, …, v*]**}

without demanding any additional explanations in view of transparency **;** at that, the**Map8** function is rather useful means, in particular, at organization of comparisons of results of the calls of functionally similar blocks/functions /modules on identical tuples of the actual arguments. While the call**Map9[*x,*** {***a,b,…,v***}***,*** {***a1,b1,…,v1***}**]** where***x*** **–** the symbol, {***a,b,c, …,v***} and {***a1,b1,c1,…,v1***} are lists of the arbitrary expressions of identical length, returns result of the following format, namely**:**

{***x*[*a, a1*],*x*[*b, b1*],*x*[*c, c1*],*x*[*d, d1*], …,*x*[*v, v1*]**}

The call **Map10[*F, x,*** {***a, b, …, v***}***,c1, c2, …, cn*]**, where***F*** **–** a symbol,***x*** and {***a, b, c, …, v***}**–** an expression and lists of expressions respectively,***c1, c2, …, cn*** **–** optional arguments, returns result of the following format, namely**:**

{***F*[*x, a, c1, c2,…*],*F*[*x, b, c1, c2,…*],*F*[*x, c, c1, c2,…*], …,*F*[*x, v, c1, c2,…*]**}

The **Map12** procedure generalizes the standard**Map** function onto a case of a nested list as its second actual argument. The call**Map12[*F,*** {{***a, b, c,…, v***}, {***a1, b1, c1,…, v1***}***, …,p, …,*** {***ap, bp, h, cp,…, vp***}}**]** where***F*** **–** a symbol, and the second argument**–** the nested list of arbitrary expressions, returns result of the following format, namely**:**

{**Map[*F,*** {***a, b, c,…, v***}**], Map[*F,*** {***a1, b1, c1,…, v1***}**], …,*F*[*p*], …, Map[*F,*** {***ap, bp, h, cp,…, vp***}**]**}

Whereas the **Map13** procedure generalizes the standard**Map** function onto case of a list of*ListList*-type as its*second* actual argument. The procedure call **Map13[*F,*** {{***a, b, c, …, v***}, {***a1, b1, c1, …, v1***}***, …,*** {***ap, bp, cp, …, vp***}}**]** where***F*** **–**

a symbol, and the second argument**–** the list of*ListList*-type of expressions, returns result of the following format, namely**:**
{***F*[*a, a1, a2, …, ap*],*F*[*b, b1, b2, …, bp*],*F*[*c, c1, c2, …, cp*], …,*F*[*v, v1, v2, …, vp*]**}

In case of an undefinite symbol ***x*** the concept of*arity* is ignored**;** meanwhile, if an actual argument***x*** defines procedure or function of the user, the call of **Map13** is returned unevaluated if*arity****x*** is other than length of sublists of***y*.** The call**Map14[*F,*** {***a, b, c, …,***

v*},*y*] where*F* –** a symbol, the second argument is a list of any expressions andy* –** an arbitrary expression, returns result of the following format, namely**:**

{***F*[*a, y*],*F*[*b, y*],*F*[*c, y*],*F*[*d, y*], …,*F*[*v, y*]**}

At that, an use at the call **Map14[F,** {***a, b, c, …, v***}***,y, t*]** of optional***4th*** actual argument**–** an arbitrary expression**–** returns result of the following format, namely**:**

{***"F*[*a, y*]", "*F*[*b, y*]", "*F*[*c, y*]", "*F*[*d, y*]", …, "*F*[*v, y*]"**} The call**Map15[*x1*,*x2, x3, …, xp, t*]** where***xj* –** symbols, and***t* –** an arbitrary admissible expression, returns result of the following format, namely**:** {***x1*[*t*],*x2*[*t*],*x3*[*t*],*x4*[*t*],*x5*[*t*], …,*xp*[*t*]**}

The call **Map16[*F,*** {***a, b, …, v***}***,c1, c2, …, cn*]** where***F* –** a symbol whereas {***a, b, c, …, v***}**–** a list of arbitrary expressions, and***c1, c2, …, cn* –** optional arguments accordingly**–** returns the list of the following format, namely**:**

{**F[*a, c1, c2, …*], F[*b, c1, c2, …*], F[*c, c1, c2, …*], …, F[*v, c1, c2, …*]**} At last, the call**Map17[*F,*** {***a –> b, c –> d, …***}**]** where***F* –** a symbol while {***a –> b, c –> d, …***}**–** a list of rules returns the list of the following format, namely**:** {**F[*a*]–> F[*b*], F[*c*]–> F[*d*], …**}

without demanding any additional explanations in view of its transparency. The seventeen means, represented above, form so-called***Map*–**group which rather significantly expands functionality of standard function**Map** of the system. From the presented information the formats of the returned results of calls of the above means are quite transparent, without demanding any additional explanations. Means of***Map***group in a number of cases allow to simplify programming of procedures and functions, significantly extending the standard**Map** function. The***AVZ_Package*** package also use these tools. It is rather expedient to programmatically process all special and erroneous situations arising in the course of calculation for that the**Mathematica** has quite enough means in the***Input*-**mode whereas at procedural processing of such situations the question is slightly more difficult. For this reason the**Try** function which represents a certain analog of***try*–**sentence of**Maple** system whose mechanism is very effective in the***Input*–**mode and at a procedural processing of special and erroneous situations when erroneous and special situations without any serious reason don**'**t lead to a procedure completion without returning the corresponding diagnostic messages has been offered. The fragment below represents source code of the**Try** function along with examples of its typical application.

In[2458] **:= G::norational = "actual argument `1` is not rational"** Out[2458]= **"**actual argument`1` is not rational**"**
In[2459]**:= G[x_] := If[Head[x] === Rational, Numerator[x]^9 +**

Denominator[x]^9, Message[G::norational, x]; Defer[G[x]]] In[2460]**:= G[42.73]**
G**::**norational**:** actual argument 42.73` is not rational Out[2460]= G[42.73]

In[2461] **:= Try[x_ /; StringQ[x], y_] := Quiet[Check[ToExpression[x],** {**y, $MessageList**}**]]**
In[2462]**:= Try["G[42/73]", "Res"]**
Out[2462]= 59 278 258 092 117 385
In[2463]**:= Try["G[42.73]", "Res"]**
Out[2463]= {**"**Res**"**, {G**::**norational}}
In[2464]**:= Try["1/0", "Error"]**

```
Out[2464]= {"Error", {Power::infy}
In[2465]:= Ag[x_Integer, y_Integer] := Module[{a = Try[ToString[x] <> "/" <>
ToString[y], "Error"]}, If[ListQ[a], If[a[[1]] === "Error", {x, y, a[[2]]}], x/y]]
In[2466]:= Ag[80, 480]
Out[2466]= 1/6
In[2467]:= Ag[80, 0]
Out[2467]= {80, 0, {Power::infy}}
```

First of all, for illustration of operating of the function **Try[*x, y*]** the message with name***"G::norational"*** that is used by simple function**G[*x*]** in case of its call on argument***x*** different from a rational number. Such call outputs this message with return of the unevaluated call*(at that,only simplifications of an expressionxcan be done)*. The**Try** function is similar to***try***–clause of**Maple,** providing processing of***x*** depending on the*messages* initiated by evaluation of***x***. Furthermore, it is necessary to note that all messages initiated by such evaluation of an expression***x***, should be activated in the current session. The call of the function has the following format, namely**:**

Try[*"x–expression",y*}}]

where the *first* argument***x*** defines an expression***x*** in string format whereas the*second* argument defines the message associated with a possible special or erroneous situation at calculation of expression***x***. In case evaluation of an expression***x*** is correct, the result of its evaluation*(for example,the procedure call)* is returned, otherwise the nested list of the format {***y***, {***Mes***}} is returned where***Mes*** defines the system message generated as a result of processing of a erroneous or special situation by the system. The function**Try** proved itself as a rather convenient means for processing of special and erroneous situations at programming of a number of applied and system problems. For the rather experienced users of the**Mathematica** system the codes of the illustrative examples and means of this section are quite transparent and of any special additional explanations don't demand.

5.1. The control branching structures and cyclic structures in the *Mathematica*system

Rather difficult algorithms of calculations and/or control algorithms *(first of all)* can't use especially*consequent* schemes, including various constructions changing consequent order of an algorithm execution depending on these or those conditions**:***conditional* and*unconditional* transitions, cycles, branchings *(the structures of such type in a number of cases are called as**control structures**)*. In particular, for the organization of the control structures of the branching type the***Math***–language of**Mathematica** has rather effective tool provided with the**If**–function having three formats of coding [28-33,60]. In a number of cases a simple**Iff** procedure from number of arguments from***1*** to***n*** that generalizes the standard**If** function is quite useful tool**;** it is very convenient at number of arguments, starting with***1***, what is convenient in cases when calls of the**Iff** function are generated in a certain procedure automatically, simplifying processing of erroneous and special situations arising by a call of such procedure on number of arguments from range***2..4***. The following fragment represents source code of the**Iff** procedure with an example. At that, it must be kept in mind that all actual arguments of***y***, since the second, are coded in string format in order to avoid their premature calculation at the call**Iff[*x*, …]** when the actual arguments are being calculated/simplified.

In[2745] **:= Iff[x_, y__ /; StringQ[y]] := Module[{a = {x, y}, b}, b = Length[a]; If[b == 1||b >= 5, Defer[Iff[x, y]], If[b === 2, If[x, ToExpression[y]],**

If[b == 3, If[x, ToExpression[y], ToExpression[a[[3]]]], If[b == 4, If[x, ToExpression[a[[2]]], ToExpression[a[[3]]], ToExpression[a[[4]]]], Null]]]]]

In[2746]**:= a = {}; For[k=1, k<=73, k++, Iff[PrimeQ[k], "AppendTo[a, k]"]]; a**
Out[2746]= {2, 3, 5,7,11, 13,17, 19, 23, 29, 31,37, 41,43, 47,53, 59, 61, 67, 71, 73}

So, the function**If** represents the most typical instrument for ensuring of the branching algorithms. In this context it should be noted that***If***–means of the **Maple** and**Mathematica** are considerably equivalent, however readability of difficult enough branching algorithms realized by***if***–offers of the**Maple** system is being perceived slightly more clearly. In particular,**Maple** allows the conditional***if***–offer of the following format, namely**:**

iflc1 then v1 elif lc2 then v2 elif lc3 then v3 elif lc4 then v4…else vk end if

where ***jthlcj***– a logical condition and***vj***– an arbitrary expression, whose sense is rather transparent and considered, for example, in books [25-27,49]. This offer is very convenient at programming of a number of conditional structures. For determination of similar structure in **Mathematica** the **IFk** procedure whose source code along with examples of usage represents the following fragment can be used, namely**:**

In[2340] **:= IFk[x__] := Module[{a = {x}, b, c = "", d = "If[", e = "]", h = {}, k = 1}, b = Length[a]; If[For[k, k <= b–1, k++, AppendTo[h, b >= 2 && ListQ[a[[k]]] && Length[a[[k]]] == 2]]; DeleteDuplicates[h] != {True}, Return[Defer[Ifk[x]]], k = 1]; For[k, k <= b–1, k++, c = c <> d <> ToString[a[[k]][[1]]] <> "," <> ToString[a[[k]][[2]]] <> ","]; c = c <> ToString[a[[b]]] <> StringMultiple[e, b–1]; ToExpression[c]]**

In[2341] **:= IFk[{a, b}, {c, d}, {g, s}, {m, n}, {q, p}, h]**
Out[2341]= If[a, b, If[c, d, If[g, s, If[m, n, If[q, p, h]]]]]
In[2342]**:= IFk[{False, b}, {False, d}, {False, s}, {True, n}, {False, p}, h]** Out[2342]= n
In[2343]**:= IFk[{False, b}, {False, d}, {False, s}, {False, n}, {g, p}, h]** Out[2343]= If[g, p, h]

In[2060] **:= IFk1[x__] := Module[{a ={x}, b, c = "", d = "If[", e = "]", h = {}, k = 1}, b = Length[a]; If[For[k, k <= b–1, k++, AppendTo[h, b >= 2 && ListQ[a[[k]]] && Length[a[[k]]] == 2]]; DeleteDuplicates[h] != {True}, Return[Defer[Ifk1[x]]], {h, k}= {{}, 1}]; If[For[k, k <= b–1, k++, AppendTo[h, a[[k]][[1]]]]; Select[h, ! MemberQ[{True, False}, #] &] != {}, Return[Defer[Ifk1[x]]], k = 1]; For[k = 1, k <= b–1, k++, c = c <> d <> ToString[a[[k]][[1]]] <> "," <> ToString[a[[k]][[2]]] <> ","]; c = c <> ToString[a[[b]]] <> StringMultiple[e, b–1]; ToExpression[c]]**

In[2061] **:= IFk1[{False, b}, {False, d}, {False, s}, {False, n}, {g, p}, h]** Out[2061]= IFk1[{False, b}, {False, d}, {False, s}, {False, n}, {g, p}, h] In[2062]**:= IFk1[{False, b}, {False, d}, {False, s}, {True, n}, {False, p}, h]** Out[2062]= n
In[2063]**:= IFk1[{a, b}, {c, d}, {g, s}, {m, n}, {q, p}, h]**
Out[2063]= IFk1[{a, b}, {c, d}, {g, s}, {m, n}, {q, p}, h]

```
In[2065]:= IFk1[{True, b}]
Out[2065]= IFk1[{True, b}]
In[2066]:= IFk1[{False, b}, Agn]
Out[2066]= Agn
```

The call of the **IF**k procedure uses any number of the actual arguments more than one**;** the arguments use the ***the arguments use the***element lists of the format {***lcj, vj***}, except the last. Whereas the last actual argument is a correct expression**;** at that, a testing of***lcj*** on Boolean type isn**'**t done. The call of the**IFk** procedure on a tuple of correct actual arguments returns the result equivalent to execution of the corresponding**Maple*****if*–**offer [25-27].
At that, the**IFk1** procedure is an useful extension of the previous procedure which unlike**IFk** allows only Boolean expressions as the actual arguments ***lcj,*** otherwise returning the unevaluated call. In the rest, the procedures**IFk** and**IFk1** are functionally identical. Thus, similarly to the***if*–**offer of**Maple** system the procedures**IFk** and**IFk1** are quite useful at programming of the branching algorithms of various types. With that said, the above procedures **IFk** and**IFk1** are provided with a quite developed mechanism of testing of the factual arguments transferred at the procedure call whose algorithm is easily seen from source code. Now, using the described approach fairly easy to program in***Math*–**language an arbitrary construction of***Maple*–**language describing the branching algorithms [28-33].
To a certain degree it is possible to refer to***If*–**constructions also the**Which–** function of the following format

Which[***lc1, w1, lc2, w2, lc3, w3,…,lc*k*, w*k**]**
that returns result of evaluation of the*first****w*j**–expression for which Boolean expression***lc*j*(j=1..k)*** accepts*True* value, for example**:**

```
In[2735] := G[x_] := Which[–Infinity <= x < 80, Sin[x], 80 <= x < 480, Cos[x], 480 <= x <= Infinity, x^2]
In[2736]:= {G[67], G[80.480], G[480], G[2014], G[–18.06]}
Out[2736]= {Sin[67], 0.361044, 230400, 4056196, 0.710041}
```

The example illustrates definition of a piecewise **–**defined function through the**Which** function. If some of the evaluated conditions***lcj*** doesn**'**t return {*True*|*False*} the function call is returned unevaluated while in case of*False* value for all conditions***lc*j*(j=1..k)*** the function call returns*Null***,** i.e. nothing. At dynamic generation of a***Which*–**object a simple**WhichN** procedure can be rather useful which allows any even number of arguments similar to the **Which–**function, otherwise returning result by unevaluated. In the rest, the **WhichN** is similar to the**Which** function**;** the following fragment represents source code of the procedure along with typical examples of its usage.

```
In[4731]:= WhichN[x__] := Module[{a = {x}, c = "Which[", d, k = 1}, d =

Length[a]; If[OddQ[d], Defer[WhichN[x]], ToExpression[For[k, k <= d, k++, c = c <> ToString[a[[k]]] <> ","]; StringTake[c, {1,–2}] <> "]"]]] In[4732]:= WhichN[a, b, c, d, f, g, h, r]
Out[4732]= Which[a, b, c, d, f, g, h, r]
In[4733]:= f = 80; WhichN[False, b, f == 80, SV, g, h, r, t]
```

Out[4733]= SV

The above procedures **IFk, IFk1** and**Which** represent a quite certain interest at programming a number of applications of various purpose, first of all, of the system character.

5.2. The cyclic control structures of the*Mathematica*system

So, one of the main cyclic structures of the system is based on the function **For** that has the following general format of coding, namely**:**
For[*a*,<*lc*>, *b*,<*Body of a cyclic construction*>]

Since the given ***a,*** the body of a construction which contains offers of***Math-*** language, with cyclic*increment* of a cyclic variable on a magnitude***b*** so long as a logical condition***(lc)*** doesn**'**t accept*True* is cyclically calculated. Simple example of usage of this function is represented below, namely**:**

In[2942] **:= For[k = 1; h = 1, k < 10000, k = k + 1, h = h^2 + 80*h + k; If[k < 5, Continue[], Print[h]; Break[]]]**
994450 659 746 015 592 932 434 074 712 430

For continuation of a ***For***cycle and exit from it the control words**Continue** and**Break** serve respectively as it very visually illustrates a simple example above. While other quite widely used means in the**Mathematica** system for organization of cyclic calculations is the**Do** function that has*five* formats of coding whose descriptions with examples can be found in books [28-33,60]. Meantime, unlike the**Maple** system the**Mathematica** system has no analog of very useful cyclic constructions of types***(1.b)*** and***(1.d)*** [27] that allow to execute cyclical calculations at subexpressions of a certain expression what provides possibility on their basis to program quite interesting constructions as illustrate simple fragments [25-27]. In this context we will represent the procedure whose call**DO[*x,y,j*]** returns the list of results of cyclic calculation of an expression***x*** on a cycle variable***j*** which accepts values from the**Op[*y*]** list. The construction in a certain relation is analog of the cyclic construction ***for_in*** for the**Maple** system [25-27,47].

In[2274] **:= DO[x_, y_, k_] := Module[{a = x, b = Op[y], c, d = 1, R = {}}, c := Length[b] + 1; While[d < c, R = Insert[R, a /. k–> b[[d]],–1]; a := x; d++]; R]**
In[2275]**:= DO[k^2 + Log[k], f[g[a, b], h[c, d, e, j, k, l]], k]**
Out[2275]= {g[a, b]^2+ Log[g[a, b]], h[c, d, e, j, k, l]^2+Log[h[c, d, e, j, k, l]]}

In our books [28 -33] the reciprocal functional equivalence of both systems is quite visually illustrated when the most important computing constructions of the**Mathematica** system with this or that efficiency are simulated by the **Maple** constructions and vice versa. Truly, in principle, it is a quite expected result because the builtin languages of both systems are*universal* and in this regard with one or the other efficiency they can program any algorithm. But in the temporary relation it is not so and at using of cyclic structures of large enough nesting level the**Maple** can have very essential advantages before **Mathematica.** For confirmation we will give a simple example of a cyclical construction programmed both in the**Maple,** and the**Mathematica.**

The results speak for themselves **–** if in**Maple*11*** the execution of a certain construction

requires*7.800* s, then**Mathematica*10*** for execution of the same construction requires already*49.358* s, i.e. approximately*6.3* times more*(the estimations have been obtained on PC**Dell Optiplex 3020,**i5–4570 3.2 GHz with 64–bit Windows 7 Professional)***.**
Furthermore, with growth of depth of nesting and range of a cycle variable at implementation of cyclic constructions this difference rather significantly grows.

```
> t := time(): for k1 to 10 do for k2 to 10 do for k3 to 10 do for k4 to 10 do for k5 to
10 do for k6 to 10 do for k7 to 10 do for k8 to 10 do 80 end do end do end do end do
end do end do end do end do: time()–t; # (Maple 11)

7.800
In[2693]:= n = 10; t = TimeUsed[]; For[k1 = 1, k1 <= n, k1++, For[k2 = 1, k2 <= n,
k2++,
For[k3 = 1, k3 <= n, k3++,
For[k4 = 1, k4 <= n, k4++,
For[k5 = 1, k5 <= n, k5++,
For[k6 = 1, k6 <= n, k6++,
For[k7 = 1, k7 <= n, k7++,

For[k8 = 1, k8 <= n, k8++, 80]]]]]]]]; TimeUsed[] –t Out[2693]= 49.358
```

Naturally, the received values are determined by the main resources of the computer however on identical resources this basic relation retains. From the given example follows the**Maple** uses more effective algorithms in the temporary relation for realization of cyclical constructions of large nesting depth and range of cycle variable, than it takes place for its main competitor
– the**Mathematica** systems even of its latest version***10.1.0.0.*** A number of interesting enough comparisons relative to estimates of time characteristics of performance of mass means of processing and calculations is represented in our books [25-27]. Of these comparisons follows, that according to time characteristics the**Maple** system in certain cases is more preferable than the **Mathematica** system what in each concrete case supposes the corresponding comparative analysis.

Among special types of cyclic control structures in the **Mathematica** system it is possible to note a series of interesting enough ones, some of them have various level of analogy with similar means of the**Maple** system [27,28-33]. However in general, means of**Mathematica** system are more preferable at generation of the nested expressions and, first of all, of pure functions that play especially essential part in problems of functional programming in the **Mathematica** system**.** At comparative consideration of the control structures of*branching* and*cycle* that are supported by both systems, two main groups have been distinguished, namely**:***basic* and*additional* resources of providing the specified control structures. So, the***if*** offer of the**Maple** system and the **If** function of the**Mathematica** system represent the most typical means of ensuring of the branching algorithms. At operating with both means a point of view has been formed, the means are being represented as substantially equivalent, however these means realized by the***if*** clause of the**Maple** for rather*complex branching* algorithms are being slightly*more* simply perceived in sence of readability. In other respects it is very difficult to give preference to any of these control means and in this relation both leading systems can quite be considered as equivalent.

Chapter 6. Problems of procedural programming in the *Mathematica*software

Procedural programming is one of basic paradigms of the**Mathematica** that in quite essential degree differs from the similar paradigm of well–known traditional procedural programming languages. The given circumstance is the cornerstone of a number of the system problems relating to a question of procedural programming in **Mathematica.** Above all, similar problems arise in the field of distinctions in realization of the above paradigms in the **Mathematica** and in the environment of traditional procedural languages. Along with it, unlike a number of*traditional* and*built-in* languages the*built-in**Math*language has no a number of useful enough means for work with procedural objects. Some such means are represented in our books [28-33] and***AVZ_Package*** package [48]. A number of the tasks connected with such means is considered in the present chapter, previously having discussed the concept of***`procedure`*** in**Mathematica** as bases of its procedural paradigm. At that, means of analysis of this section concern only the user procedures and functions because definitions of all system functions*(unlike,say,from the **Maple**)* from the user are hidden, i.e. are inaccessible by standard means of the**Mathematica** system.

6.1. Definition of procedures in the*Mathematica*software

Procedures in the**Mathematica** system formally represent functional objects of following two simple formats, namely**:**
M*[*x_/*;*Testx*,*y_/*;*Testy*, …]** {:= | =}**Module[**{locals***}**,*Procedure Body*]**
B*[*x_/*;*Testx*,*y_/*;*Testy*, …]** {:= | =}**Block[**{locals***}**,*Procedure Body*]**

i.e., the procedures of both types represent functions from two arguments **–** *body* of a procedure*(**Body**)* and*local variables (**locals**)***.** Local variables**–** the list of names, perhaps, with the initial values which are attributed to them. These variables have local character concerning procedure, i.e. their values aren**'**t crossed with values of the symbols of the same name outside of the procedure. All other variables in procedure have global character, dividing area of variables of the**Mathematica** current session.
Thus, in definition of procedures it is possible to distinguish five following component, namely**:**

***– procedure name** (**M**in the first procedure definition);*
***– procedure heading**(**M*[*x_/*;*Testx*,*y_/*;*Testy*, …]**in the both procedures definitions);*
–procedural brackets (**Module[…]***or***Block[…]**);
–local variables**(list of local variables* {locals***}*;can be empty);*
– procedure body;*can be empty.*

Above all, it should be noted the following very important circumstance. If in the traditional programming languages the identification of an arbitrary procedure/function is made according to its*name***,** in case of***Math***language identification is made according to its*heading*. The circumstance is caused by that the definition of a procedure/function in***Math***language is made by the manner different from traditional [28-33]. Simultaneous existence of the procedures/functions of the same name with various headings in the given situation is admissible as it illustrates the following fragment, namely:

```
In[2434] := M[x_, y_] := Module[{}, x + y]; M[x_] := Module[{}, x^2]; M[y_] :=
Module[{}, y^3]; M[x___] := Module[{}, {x}]
In[2435]:= Definition[M]
Out[2435]= M[x_, y_]:= Module[{}, x+ y]
M[y_]:= Module[{}, y^3]
M[x___]:= Module[{}, {x}]
In[2436]:= {M[480, 80], M[80], M[42, 47, 67, 25, 18]}
Out[2436]= {560, 512000, {42, 47, 67, 25, 18}}
In[2437]:= G[x_Integer] := Module[{}, x]; G[x_] := Module[{}, x^2]; G[480]
Out[2437]= 480
```

At the call of a procedure/function of the same name from the list is chosen the one, whose formal arguments of the heading correspond to the factual arguments of the call, otherwise the call is returned by***unevaluated,*** except for*simplifications* of the actual arguments according to the standard system agreements. Moreover, at compliance of formal arguments of heading with the actual ones a procedure**x** is caused, whose definition is above in the list returned at the**Definition[*x*]** call; in particular, whose definition has been calculated in the**Mathematica** current session by the first.
Further is being quite often mentioned about return of result of the call of a function/procedure by unevaluated, it concerns both the standard system means, and the user means. In any case, the call of a procedure/function on an*inadmissible* tuple of actual arguments is returned by unevaluated, except for standard simplifications of the actual arguments. In this connection the **UnevaluatedQ** procedure providing testing of a certain procedure/function regarding of return of its call unevaluated on a concrete tuple of the factual arguments has been programmed. The call**UnevaluatedQ[*F*,*x*]** returns*True* if the call**F[*x*]** is returned unevaluated, and*False* otherwise; in addition, on an erroneous call**F[*x*]*"ErrorInNumArgs"*** is returned. The fragment below represents source code of the **UnevaluatedQ** procedure with examples of its usage. Procedure represents a certain interest for program processing of results of calls of procedures and functions.

```
In[3246] := UnevaluatedQ[F_ /; SymbolQ[F], x___] :=
Module[{a = Quiet[Check[F[x], "e", F::argx]]}, If[a === "e", "ErrorInNumArgs",
If[ToString1[a]===ToString[F]<>"[" <> If[{x}== {}, "",
ListStrToStr[Map[ToString1, {x}]] <> "]"], True, False]]]

In[3247] := {UnevaluatedQ[F, x, y, z], UnevaluatedQ[Sin, x, y, z]} Out[3247]= {True,
"ErrorInArgs"}
In[3248]:= {UnevaluatedQ[Sin, 48080], UnevaluatedQ[Sin, 480.80],

UnevaluatedQ[Sin]}
Out[3248]= {True, False, "ErrorInNumArgs"}
```

Meanwhile, the standard **Definition** function in the case of the procedures/ functions of the same name in a number of cases is of little use for solution of tasks which are connected with processing of definitions of such objects. Above all, it concerns the procedures whose definitions are located in the user packages loaded into the current session of**Mathematica** as illustrates a simple example of receiving definition of the**SystemQ** procedure [48]:

```
In[3247]:= Definition["SystemQ"]
Out[3247]= SystemQ[AladjevProcedures`SystemQ`S_]:=

If[Off[ "Definition"::"ssle"]; !
ToString["Definition"[AladjevProcedures`SystemQ`S]]=== Null&&
SysFuncQ1[AladjevProcedures`SystemQ`S], On["Definition"::"ssle"]; True,
On["Definition"::"ssle"]; False]
```

The call**Definition[*x*]** of the standard function in a number of cases returns the definition of some object*x* with the context corresponding to it what at quite large definitions becomes bad foreseeable and less acceptable for the subsequent program processing as enough visually illustrates the previous example. Moreover, the name of object or its string format also can act as an actual argument. For elimination of this shortcoming we defined a number of means allowing to obtain definitions of procedures/functions in a certain optimized format. As such tools it is possible to note such as**DefOptimum, Definition1, Definition2, Definition3, Definition4, DefFunc, DefFunc1, DefFunc2, DefFunc3** and**DefOpt.** These means along with some others are presented in books [28-33] and included in the***AVZ_Package*** package [48]. The following fragment represents source codes of the most used of them.

```
In[2470]:= Definition2[x_ /; SameQ[SymbolQ[x], HowAct[x]]] := Module[{a, b =
Attributes[x], c},

If[SystemQ[x], Return[ {"System", Attributes[x]}], Off[Part::partw]];
ClearAttributes[x, b]; Quiet[a = ToString[InputForm[Definition[x]]];
Mapp[SetAttributes, {Rule, StringJoin}, Listable]; c = StringReplace[a,
Flatten[{Rule[StringJoin[Contexts1[], ToString[x] <> "`"], ""]}]]; c = StringSplit[c,
"\n \n"]; Mapp[ClearAttributes, {Rule, StringJoin}, Listable]; SetAttributes[x, b]; a
= AppendTo[c, b]; If[SameQ[a[[1]], "Null"] && a[[2]] == {}, On[Part::partw];
{"Undefined", Attributes[x]}, If[SameQ[a[[1]], "Null"] && a[[2]] != {}&& !
SystemQ[x], On[Part::partw]; {"Undefined", Attributes[x]}, If[SameQ[a[[1]],
"Null"] && a[[2]] != {}&& a[[2]] != {}, On[Part::partw]; {"System", Attributes[x]},
On[Part::partw]; a]]]]]

In[2471]:= Definition2[SystemQ]
Out[2471]= {"SystemQ[S_]:= If[Off[MessageName[Definition, "ssle"]];

! ToString[Definition[S]]=== Null&& SysFuncQ1[S], On[MessageName[Definition,
"ssle"]]; True, On[MessageName[Definition,"ssle"]]; False]", {}} In[2472]:=
Definition2[Tan]

Out[2472]= {"System", {Listable, NumericFunction, Protected}}
In[2473]:= Definition2[(a + b)/(c + d)]
Out[2473]= Definition2[(a+ b)/(c+ d)]
In[2502]:= Definition3[x_ /; SymbolQ[x], y_ /; ! HowAct[y]] := Module[{a =
Attributes[x], b = Definition2[x]},

If[b[[1]] == "System", y = x; {"System", a}, b = Definition2[x][[1 ;;–2]];
ClearAttributes[x, a]; If[BlockFuncModQ [x, y], ToExpression[b]; SetAttributes[x,
a]; Definition[x], SetAttributes[x, a]; Definition[x]]]]
```

In[2503]:= **Definition3[SystemQ, y]**
Out[2503]= SystemQ[S_]:= If[Off["Definition"::"ssle"];
! ToString["Definition"[S]]=== Null**&&** SysFuncQ1[S]**,**

On[**"**Definition**"::"**ssle"]**;** True**,** On["Definition**"::"**ssle"]**;** False] In[2504]**:= y**
Out[2504]= **"**Function**"**
In[2505]**:= Definition3[Sin, z]**
Out[2505]= Attributes[Sin]= {Listable**,** NumericFunction**,** Protected} In[2506]**:= z**
Out[2506]= Sin

In[2598] **:= Definition4[x_ /; StringQ[x]] := Module[**{**a**}**,**
a = Quiet[Check[Select[StringSplit[ToString[InputForm[Quiet[Definition[x]]]], "\n"], # != " " && # != x &], $Failed]]; If[a === $Failed, $Failed, If[SuffPref[a[[1]], "Attributes[", 1], a = AppendTo[a[[2 ;;–1]], a[[1]]]]]; If[Length[a] != 1, a, a[[1]]]]]

In[2599]**:= W = 80; G := 480; Map[Definition4,** {**"W", "G", "72", "a + b", "If"**}**]**
Out[2599]= {"W= 80**", "**G:= 480**",** $Failed**,** $Failed**, "**Attributes[If]= {HoldRest**,** Protected}**"**}
In[2600]**:= A[x_] := Module[**{**a=90**}**, x+a]; A[x_, y_] := Module[**{**a=6**}**, x+y+a];**

A[x_, y_List] := Block[{}**,** {**x, y**}**]; A[x_Integer] := Module[**{**a = 480**}**, x + a]; A :=** {**a, b, c, d, h**}**; SetAttributes[A,** {**Flat, Listable, Protected**}**];** In[2601]**:= Definition4["A"]**
Out[2601]= {"A:= {a**,** b**,** c**,** d**,** h}**", "**A[x_Integer]:= Module[{a= 480}**,** x+ a]**",**

"A[x_]**:=** Module[{a= 90}**,** x+ a]**", "**A[x_**,** y_List]**:=** Block[{}**,** {x**,** y}]**", "**A[x_**,** y_]**:=** Module[{a= 6}**,** x+ y+ a]**", "**Attributes[A]=

{Flat **,** Listable**,** Protected}**"**}
A number of functional means of***Math*–**language as the actual arguments assume only objects of the types {*Symbol***,***String***,***HoldPattern*[*Symbol*]} what in some cases is quite inconvenient at programming of problems of different purpose. In particular, the**DefinitionDefinition** 33]. For the purpose of expansion of the**Definition** function on the types, different from the mentioned ones, the**Definition1** procedure can be used, whose call**Definition1[*x*]** returns definition of an object***x*** in string format, **"***Null***"** if***x*** isn**'**t defined, otherwise**$**Failed is returned. A fragment in [28,33] represents the procedure code with typical examples of its application from which certain advantages of the**Definition1** concerning the**Definition** are quite visually visible. The**Definition1** procedure has been realized with use of our**ToString1** procedure [28] that unlike the standard**ToString** function provides correct converting of expressions in string format. The**Definition1** procedure processes the main special and erroneous situations. Meanwhile, the**Definition1** procedure doesn**'**t rid the returned definitions from contexts and is correct only for objects with unique names. Moreover, in case of the multiple contexts the**Definition1** procedure call returns definitions with a context that answers the last user package loaded into the current session. On system functions, the**Definition1** procedure call returns**"***Null***".** As an expansion of the**Definition1** procedure the**Definition2** procedure represented by the previous fragment can serve. The given procedure uses our means**Contexts1, HowAct, Mapp, SymbolQ** and**SystemQ** considered in the present book and in [28-33]. These means are rather simple and are used in our means enough widely [48].

Unlike previous procedure **Definition1,** the**Definition2** procedure rids the returned

definitions from contexts, and is correct for program objects with unique names. The**Definition2** call on system functions returns the nested list, whose first element**–** ***"System"*, ** whereas the second element**–** the list of attributes ascribed to a factual argument. On a function or procedure of the user***x*** the call**Definition2[*x*]** also returns the nested list, whose first element
– the*optimized* definition of***x****(in the sense of absence of contexts in it)***,** whereas the second element**–** the list of attributes ascribed to***x;*** in their absence the empty list acts as the second element of the returned list. In the case of*False* value on a test ascribed to formal argument***x*,** the call**Definition2[*x*]** will be returned*unevaluated.* Analogously to the previous procedure, the procedure **Definition2** processes the main*special* and*erroneous* situations. In addition, **Definition1** and**Definition2** return definitions of objects in string format. The call**Definition3[*x*, *y*]** returns the optimum definition of a procedure or a function*x*, while through the second argument***y*** ***–****an undefined variable***–** the type of***x*** in a context {***"****Procedure****"*, *"****Function****"*, *"****Procedure***&***Function"***} is returned if***x*** is a procedure or function, on system functions the procedure call returns the list {***"****Function****"*,** {*Attributes*}} while thru the second argument ***y*** the*first* argument is returned**;** at inadmissibility of the*first* argument***x*** the call is returned unevaluated, i.e.**Definition[*x*].**

The call **Definition4[*x*]** in a convenient format returns the definition of an object***x*** whose name is coded in the string format, namely**:*(1)*** on a system function***x*** its attributes are returned**,*(2)*** on the user block, function, module the call returns the definition of object***x*** in string format with the attributes, options and/or values by default for formal arguments ascribed to it*(if such are available)***,** ***(3)*** the call returns the definition of an object***x*** in string format for assignments by operators {**":="**, **"="**}**,** and***(4)*** in other cases the procedure call returns**$**Failed. The procedure has a number of interesting appendices at programming of various system appendices.
The following**DefOpt** procedure represents a quite certain interest that in a number of cases is more acceptable than the**Definition** function along with our procedures**DefFunc, DefFunc1, DefFunc2** and**DefFunc3** considered in [28-33,48] that are also intended for obtaining definitions of procedures and functions in the convenient form acceptable for processing. The following fragment represents source code of the procedure with examples of its use.

In[2342] **:= DefOpt[x_ /; StringQ[x]] := Module[{a = If[SymbolQ[x], If[SystemQ[x], b = "Null", ToString1[Definition[x]], "Null"]], b, c}, If[! SameQ[a, "Null"], b = Quiet[Context[x]]]; If[! Quiet[ContextQ[b]], "Null",**

c = StringReplace[a, b <> x <> "`"–> ""]; ToExpression[c]; c]]

In[2343] **:= DefOpt["SystemQ"]**
Out[2343]= SystemQ[S_]**:=** If[Off[MessageName[Definition**,** **"**ssle**"**]]**;** **!** ToString[Definition[S]]=== Null**&&** SysFuncQ1[S]**,** On[MessageName[Definition**,** **"**ssle**"**]]**;** True**,** On[MessageName[Definition**,"**ssle**"**]]**;** False]

In[2344] **:= DefFunc[$TypeProc]**
Out[2344]= Attributes[**$**Failed]= {HoldAll**,** Protected}
In[2345]**:= DefOpt["$TypeProc"]**
Out[2345]= **"**$TypeProc**:=** CheckAbort[If[$Art24$Kr17$ **=**

Select[{Stack[Module] , Stack[Block], Stack[DynamicModule]}, #1!= {}&];
If[$Art24$Kr17$ == {}, Clear[$Art24$Kr17$]; Abort[], $Art24$Kr17$ =
ToString[$Art24$Kr17$[[1]][[1]]]]; SuffPref[$Art24$Kr17$, "Block[{", 1],
Clear[$Art24$Kr17$]; "Block", If[SuffPref[$Art24$Kr17$,
"Module[{", 1]&& !StringFreeQ[$Art24$Kr17$, "DynamicModule"],
Clear[$Art24$Kr17$]; "DynamicModule", Clear[$Art24$Kr17$]; "Module"]], $Failed]”

In[2346]:= **Map[DefOpt,** {**“If”, “Sin”, “Goto”, “a + b”, “80”, 480**}**]** Out[2346]= {Null, Null, Null, Null, Null, DefOpt[480]}

On the other hand, our some procedures are unsuitable in case of necessity of receiving definitions of a number of procedural variables, in particular, **$TypeProc** as some illustrate examples in [33]. And only the procedure call **DefOpt[*x*]** returns definition of an arbitrary object***x*** in an optimum format irrespective of type of the user object***x*.** At that, the call**DefOpt[*x*]** not only returns an optimum form of definition of an object***x*** in string format, but also evaluates it in the current session what in a number of cases is useful enough; at the procedure call the name of the object***x*** is coded in the string format; while on the system functions and other string expressions the call **DefOpt[*x*]** returns***”Null“*.** At the same time it must be kept in mind that the **DefOpt** is inapplicable to the procedures / functions of the same name, i.e. having several definitions with different headings. The previous fragment represents source code of the**DefOpt** procedure with examples of its usage.

The **OptDefinition** procedure is an interesting enough modification of the previous procedure; its source code with examples of usage represents the following fragment. The call **OptDefinition[*x*]** returns the definition of a procedure or a function***x*** optimized in the above sense i.e. without context associated with the user package containing the procedure or function***x*.**

In[3298] **:= OptDefinition[x_ /; Quiet[ProcQ[x] || FunctionQ[x]]] := Module[**{**c = $Packages, a, b, d, h = Definition2[x]**}, {**a, b**}**=** {**h[[1 ;;–2]], h[[–1]]**}**;
ClearAllAttributes[x]; d = Map[StringJoin[#, ToString[x] <> “`”] &, c];**

**ToExpression[Map[StringReplace[#, GenRules[d, ””]] &, a]]; SetAttributes[x, b];
Definition[x]]**

In[3299] **:= SetAttributes[ToString1,** {**Listable, Protected**}**];
Definition[ToString1]**
Out[3299]= Attributes[ToString1]= {Listable, Protected}
ToString1[AladjevProcedures`ToString1`x_]:=
Module[{AladjevProcedures`ToString1`a= “$Art23Kr15$.txt“,
AladjevProcedures`ToString1`b= ””, AladjevProcedures`ToString1`c,
AladjevProcedures`ToString1`k= 1},
Write[AladjevProcedures`ToString1`a,
AladjevProcedures`ToString1`x];
Close[AladjevProcedures`ToString1`a];
For[AladjevProcedures`ToString1`k,
AladjevProcedures`ToString1`k< \[Infinity], AladjevProcedures`ToString1`k++,
AladjevProcedures`ToString1`c=

Read[AladjevProcedures`ToString1`a, String]; If[AladjevProcedures`ToString1`c===
EndOfFile, Return[DeleteFile[
Close[AladjevProcedures`ToString1`a]]; AladjevProcedures`ToString1`b],
AladjevProcedures`ToString1`b= AladjevProcedures`ToString1`b<>
StrDelEnds[AladjevProcedures`ToString1`c, " ", 1]]]]
In[3300]:= **OptDefinition[ToString1]**
Out[3300]= Attributes[ToString1]= {Listable, Protected}
ToString1[x_]:= Module[{a= "$Art25Kr18$.txt", b= "", c, k= 1}, Write[a, x]; Close[a];
For[k, k< ∞, k++, c= Read[a, String]; If[c=== EndOfFile, Return[DeleteFile[Close[a]];
b], b= b<> StrDelEnds[c, " ", 1]]]]

It is necessary to pay attention to use of the **GenRules** procedure providing generation of the list of transformation rules for providing of replacements in a string definition of an object***x.*** In a number of cases such approach is a rather effective at strings processing. The**DefOptimum** procedure realized in a different manner is full analog of the previous procedure, whose call**DefOptimum[*x*]** returns the definition of a function or procedure***x*** optimized in the respect that it doesn't contain a*context* of the user package containing definition of the procedure/function ***x.*** The following fragment represents source code of this procedure with a typical example of its usage.

In[2245] **:= SetAttributes[OptDefinition, {Listable, Protected}];
Definition[OptDefinition]**
Out[2245]= Attributes[OptDefinition]= {Listable, Protected}
OptDefinition[x_ /; ProcQ[x]|| FunctionQ[x]]:= Module[{a= Definition2[x][[1;;–2]], b=
Definition2[x][[–1]], AladjevProcedures`OptDefinition`c= $Packages,
AladjevProcedures`OptDefinition`d,
AladjevProcedures`OptDefinition`h},
ClearAllAttributes[ToString1];
AladjevProcedures`OptDefinition`d=(#1<> (ToString[x]<> "`")&)/@
AladjevProcedures`OptDefinition`c;
ToExpression[(StringReplace[#1,
GenRules[AladjevProcedures`OptDefinition`d, ""]]&)/@ a]; SetAttributes[x, b];
"Definition"[x]]
In[2246]:= **DefOptimum[x_ /; Quiet[ProcQ[x] || FunctionQ[x]]] := Module[{a, c, k = 1, b = "Art$Kr.txt", d = Context[x], f = Attributes[x]}, ClearAttributes[x, f]; Save[a = ToString[x], x]; For[k, k < Infinity, k++, c = Read[a, String]; If[SameQ[c, EndOfFile], Break[], Write[b, StringReplace[c, d <> ToString[x] <> "`"–> ""]]]]; Map[Close, {a, b}]; Get[b]; Map[DeleteFile, {a, b}]; SetAttributes[x, f]; Definition[x]]**
In[2247]:= **DefOptimum[OptDefinition]**
Out[2247]= Attributes[OptDefinition]= {Listable, Protected}
OptDefinition[x_ /; Quiet[ProcQ[x]|| FunctionQ[x]]]:= Module[{c= $Packages, a, b, d, h=
Definition2[x]}, {a, b}= {h[[1;;–2]], h[[–1]]}; ClearAllAttributes[x]; d= (#1<>
(ToString[x]<> "`")&)/@ c; ToExpression[(StringReplace[#1, GenRules[d, ""]]&)/@ a];
SetAttributes[x, b]; "Definition"[x]]

In[2500]:= **DefOpt1[x_] := Module[{a = ToString[x], b, c},
If[! SymbolQ[a], $Failed, If[SystemQ[x], x,**

If[ProcQ[a]||FunctionQ[a], b = Attributes[x]; ClearAttributes[x, b]; c = StringReplace[ToString1[Definition[x]], Context[a] <> a <> "`"–> ""]; SetAttributes[x, b]; c, $Failed]]]]

In[2501] **:= DefOpt1[StrStr]**
Out[2501]= **"**StrStr[x_]**:=** If[StringQ[x], "<>x<>"**,** ToString[x]]**"** In[2502]**:=**
Map[DefOpt1, {**a + b, 72, Sin**}**]**
Out[2502]= {$Failed**,** $Failed**,** Sin}

The algorithm of the above **DefOptimum** procedure is based on saving of the current definition of a block/function/module***x*** in an*ASCII* format file with the subsequent its converting into the*txt***–**datafile containing definition of the object***x*** without occurrences of a package context in which definition of this object is located. Whereupon the result datafile is loaded by means of the**Get** function into the current session of the**Mathematica** with return of the optimized definition of the object***x*.**

Meanwhile, the last **DefOpt1** procedure of the previous fragment is seemed as effective enough for receiving of definition of the user procedure/function in optimized format in the above sense, i.e. without context. The procedure call**DefOpt1[*x*]** on a system function***x*** returns its name, on an user function or procedure returns its optimized code in string format whereas on other values of argument***x*****$**Failed is returned. The previous fragment represents source code of the procedure**DefOpt1** along with examples of its usage. For a number of appendices, including appendices of system character, the standard**Definition** function seems as important enough means whose call **Definition[*x*]** returns the definition of an object***x*** with attributes ascribed to it**;** in the absence of definition the*Null***,** i.e. nothing**,** or the ascribed attributes to an undefined symbol***x*** is returned, namely**:**

Attributes[*x*] =*The list of attributes ascribed to a symbol x* As very visually illustrate the following simple examples, namely**:**

In[2839]**:= SetAttributes[h, Listable]; Definition[h]**
Out[2839]= Attributes[h]= {Listable}

In[2840] **:= Definition[Sin]**
Out[2840]= Attributes[Sin]= {Listable**,** NumericFunction**,** Protected} Meanwhile, on the other hand, many problems of processing of objects are based strictly speaking on their definitions in their pure form. Therefore the allotment of definition of an arbitrary object***x*** in pure form can be provided, in particular, by means of***2*** mechanisms whose essence is explained by the examples in [28-33] by means of the procedures**Def** and**Def1;** definition of the procedure**Def1** gives the following example, namely**:**

In[2526]**:= Def1[x_ /; StringQ[x]] := Module[**{**a**}**, If[! SymbolQ[x] || SystemQ[x], $Failed, a = Definition2[x][[1 ;;–2]]; If[Length[a] == 1, a[[1]], a]]]**

In[2527] **:= B[x_] := x; B[x_, y_] := Module[**{**a, b, c**}**, x + y];**
B[x_ /; IntegerQ[x]] := Block[{**a, b, c, d**}**, x]**
In[2528]**:= SetAttributes[B,** {**Protected**}**]; Attributes[B]**
Out[2528]= {Protected}
In[2529]**:= Def1["B"]**
Out[2529]= {**"**B[x_ /; IntegerQ[x]]**:=** Block[{a**,** b**,** c}**,** x]**",**

```
"B[x_]:= x", "B[x_, y_]:= Module[{a, b, c, d}, x+ y]"}
In[2530]:= Definition[B]
Out[2530]= Attributes[B]= {Protected}
B[x_ /; IntegerQ[x]]:= Block[{}, x]}
B[x_]:= x
B[x_, y_]:= Module[{}, x+ y]
```

In event of an object ***x*** of the same name the procedure call**Def1[*x*]** returns the list of the optimized definitions of the object***x*** in string format without the attributes ascribed to it. If***x*** defines an unique name, the call returns the optimized definition of the object***x*** in string format without the attributes ascribed to it. The name of an object***x*** is given in string format; in addition, on unacceptable values of argument***x*****$**Failed is returned.
An extension of the standard**Attributes** function is represented by simple procedure, whose call**Attributes1[*x, y, z, t, …*]** unlike standard function on objects***x, y, z, t, …,*** that differ from admissible ones, returns the empty list, i.e. {}, without output of any error messages what in a number of cases more preferably from standpoint of processing of erroneous situations. While on admissible objects***x, y, z, …*** the call**Attributes1[*x, y, z, …*]** returns the list of the attributes ascribed to objects***x, y, z, …*** The following fragment represents source code of the procedure along with typical examples of its usage.

```
In[2796]:= Attributes1[x__] := Module[{a=
Map[Quiet[Check[Attributes[#], {}]] &, {x}]}, If[Length[a] == 1, a[[1]], a]]

In[2797] := L := {42, 47, 67, 25, 18}; SetAttributes[L, {Flat, Protected, Listable}]
In[2798]:= Attributes1[L[[5]], Sin, ProcQ]
Out[2798]= {{}, {Listable, NumericFunction, Protected}, {}}
In[2799]:= Attributes1[72, a + b, Attributes1, While, If]
Out[2799]= {{}, {}, {}, {HoldAll, Protected}, {HoldRest, Protected}}
```

The **Attributes1** procedure is a rather useful tool in a number of appendices. Means of processing of the attributes specific to procedures/functions in the form of procedures**AttributesH** and**DefAttributesH** are presented below.

As it was noted above, the strict differentiation of the blocks, functions and modules in the**Mathematica** is carried out not by means of their names as it is accepted in the majority of known programming languages and systems, but by means of their*headings.* For this reason in a number of cases of the advanced procedural programming, the important enough problem of the organization of mechanisms of the differentiated processing of such objects on the basis of their headings arises. A number of such means is presented in the present book, here we will determine two means ensuring work with attributes of objects on the basis of their headings. The following fragment represents source codes of***2*** means**DefAttributesH** and**AttributesH** along with typical examples of their usage.
The call**DefAttributesH[*x, y, z, p, h, …*]** returns*Null,* i.e. nothing, assigning {***y*= *"Set"***} or deleting {***y* = *"Clear"***} the attributes determined by arguments {***z,p, h, …***} for an object with heading***x.*** Whereas in attempt of assigning or deleting of an attribute, nonexistent in the current version of the system, the procedure call returns the list whose***1st*** element is**$Failed,** whereas the***2nd*** element– the list of the expressions different from the current attributes. At that, the call**AttributesH[*x*]** returns the list of attributes ascribed to an object

with heading*x.* An useful function whose call**ClearAllAttributes[*x, y, z, …*]** returns*Null*, i.e. nothing, canceling all attributes ascribed to the symbols*x, y, z, …* completes the given fragment.

In[2880]:= **DefAttributesH[x_ /; HeadingQ[x], y_ /; MemberQ[{"Set", "Clear"}, y], z___] :=**

Module[{a}, If[AttributesQ[{z}, a = Unique[g]], ToExpression[y <> "Attributes[" <> HeadName[x] <> ", " <> ToString[{z}] <> "]"], {$Failed, a}]]

In[2881] := **M[x__] := Module[{}, {x}]; M[x__, y_] := Module[{}, {x}]; M[x___, y_, z_] := x + y + z**
In[2882]:= **DefAttributesH["M[x___, y_, z_]", "Set", Flat, Protected, Listable]**
In[2883]:= **Attributes[M]**
Out[2883]= {Flat, Listable, Protected}
In[2884]:= **DefAttributesH["M[x___, y_, z_]", "Set", AvzAgn]**
Out[2884]= {$Failed, {AvzAgn}}

In[2930]:=**AttributesH[x_ /; HeadingQ[x]] := Attributes1[Symbol[HeadName[x]]]** In[2972]:= **AttributesH["M[x___, y_, z_]"]**
Out[2972]= {Flat, Listable, Protected}
In[2973]:= **ClearAllAttributes[x__] := Map[Quiet[ClearAttributes[#, Attributes[#]];] &, {x}][[1]]**

In[2974] := **SetAttributes[G, {Flat, Protected}]; SetAttributes[V, {Protected}]**
In[2975]:= **Map[Attributes, {G, V}]**
Out[2975]= {{Flat, Protected}, {Protected}}
In[2976]:= **ClearAllAttributes[G, V]; Attributes[G]**
Out[2976]= {}

The represented means equally with **Attributes1** and**SetAttributes1** [33,48] of work with attributes that are most actual for objects of the type {*Function, Block,Module*}, in some cases are rather useful. At that, these means can be used quite effectively at programming and other means of*different* purpose, first of all, of means of the system character.

The mechanism of the **Mathematica** attributes is quite effective tool both for protection of objects against modifications, and for management of a mode of processing of arguments at calls of blocks, functions and modules. So, by means of assignment to a procedure or function of the***Listable*** attribute can be specified that this procedure or function has to be applied automatically to all actual arguments as for a list elements. In the following fragment the simple procedure is presented, whose call**CallListable[*x, y*]** returns the list of values**Map[*x*,Flatten [{*y*}]],** where***x*** **–** a block, function or module from one formal argument, and***y*** **–** the list or sequence of the actual arguments that can be and empty. The fragment represents source code and the most typical examples of usage of the**CallListable** procedure.

In[2977]:= **ToString[{a, b, c + d, 72, x*y, (m + n)*Sin[p–t]}]**
Out[2977]= "{a, b, c+ d, 72, x y, (m+ n) Sin[p– t]}"
In[2978]:= **CallListable[x_ /; SystemQ[x] || BlockFuncModQ[x], y___] := Module[{a = Attributes[x]}, If[MemberQ[a, Listable], x[Flatten[{y}]], SetAttributes[x, Listable]; {x[Flatten[{y}]], ClearAttributes[x, Listable]}[[1]]]]**

In[2979] **:= CallListable[ToString, {a, b, c + d, 80, x*y, (m + n)*Sin[p–t]}]** Out[2979]= {"a", "b", "c+ d", "80", "x y", "(m+ n) Sin[p– t]"}
In[2980]**:= CallListable[ToString, a, b, c + d, 480, x*y, (m + n)*Sin[p–t]]** Out[2980]= {"a", "b", "c+ d", "480", "x y", "(m+ n) Sin[p– t]"}
In[2981]**:= CallListable[ToString]**
Out[2981]= {}

The approach used by the**CallListable** procedure is quite effective and can be used in a number of the appendices programmed in the**Mathematica.**

In conclusion of this section some useful means for receiving the optimized definitions of procedures/functions are in addition represented. So, the call **DefFunc[*x*]** provides return of the*optimized* definition of an***x***–object whose definition is located in the user package or***nb***-document and which has been loaded into the current session. At that, the name***x*** should define an object without any attributes and options**;** otherwise the erroneous situation arises. The fragment below represents source code of the**DefFunc** procedure with typical examples of its usage.

In[2461]**:= DefFunc[x_ /; SymbolQ[x] || StringQ[x]] :=**
Module[{a = GenRules[Mapp[StringJoin, {"Global`", Context[x]},

ToString[x] <> "`"], ""], b = StringSplit[ToString[InputForm[Definition[x]]], "\n \n"]}, ToExpression[Map[StringReplace[#, a] &, b]]; Definition[x]] In[2462]**:=**
Definition[ListListQ]
Out[2462]= ListListQ[AladjevProcedures`ListListQ`L_]**:=**

If[AladjevProcedures`ListListQ`L**!=** {}**&&**
ListQ[AladjevProcedures`ListListQ`L]**&&**
Length[Select[AladjevProcedures`ListListQ`L**,** ListQ[#1]**&&** Length[#1]**==**
Length[AladjevProcedures`ListListQ`L[[1]]]**&**]]**==**

Length[AladjevProcedures `ListListQ`L]**,** True**,** False] In[2463]**:= DefFunc[ListListQ]**
Out[2463]= ListListQ[L_]**:=** If[L**!=** {}**&&** ListQ[L]**&&**

Length[Select[L**,** ListQ[#1]**&&** Length[#1]**==** Length[L[[1]]]**&**]]**==** Length[L]**,** True**,** False]

Naturally, the standard **Definition** function also is suitable for receiving of definition of an object activated in the current session, but in case of a***m***file or a***nb***–document exists an essential distinction as is well illustrated by the return of definition of the**ListListQ** function from***AVZ_Package*** package [48] by means of both the**Definition** function, and our**DefFunc** procedure. In the second case the obtained definition is essentially more readably, first of all, for large source codes of procedures and functions. With procedures **DefFunc1, DefFunc2** and**DefFunc3** which are quite useful versions of the above**DefFunc** procedure, the interested reader can familiarize oneself in [28-33]**;** the means are presented and in our package [48]. These means also are functionally adjoined by the **ToDefOptPF** procedure and the **OptRes** function [28,48]. Meanwhile, the**ToDefOptPF** procedure is inefficient for a case when the user packages with identical contexts have been loaded into the current session [33]. We for the similar purposes widely use the above **Definition2** procedure.

Withal, having provided loading of the user package for instance ***UPackage*** by the call**LoadMyPackage[*"… \UPackage.mx", Context*],** all definitions containing in it will be in the optimized format, i.e. they will not contain the context associated with the***UPackage*** package. At that, processing of means of a package loaded thus will be significantly simpler. The**LoadMyPackage** procedure is considered in appropriate section. In the subsequent sections of the presented book the means of manipulation with the main components of definitions of the user procedures and functions are considered.

6.2. Definition of the user functions and pure functions in software of the*Mathematica*system

First of all, we will notice that so **–**called***functional programming*** isn**'**t any discovery of**Mathematica** system, and goes back to a number of software means that appeared long before the above system. In this regard we quite pertinently have focused slightly more in details the attention on the*concept* of functional programming in a historical context [30-33]. Whereas here we only will note certain moments characterizing specifics of the paradigm of functional programming. We will note only that the foundation of*functional* programming has been laid approximately at the same time, as*imperative* programming that is the most widespread now, i.e. in the***30****th* years of the last century.***A. Church****(USA)***–** the author of***λ*–**calculus and one of founders of the concept of Homogeneous structures*(Cellular Automata)* in connection with his works in the field of infinite abstract automata and mathematical logic along with***H. Curry****(England)* and***M. Schönfinkel****(Germany)* that have developed the mathematical theory of*combinators,* with good reason can be considered as the main founders of mathematical foundation of functional programming. At that,*functional* programming languages, especially*purely* functional ones such as*Hope* and*Rex,* have largely been used in academical circles rather than in commercial software development. While prominent *functional* programming languages such as*Lisp* have been used in industrial and commercial applications. Today, functional programming paradigm is also supported in some*domain-specific* programming languages for example by***Math***language of the**Mathematica** system. From rather large number of languages of functional programming it is possible to note the following languages which exerted a great influence on progress in this field, namely**:** ***Lisp, Scheme, ISWIM,*** family***ML, Miranda, Haskell, Clean,*** etc. [33]. By and large, if the*imperative* languages are based on operations of assignment and cycle**,** the*functional* languages on recursions. From advantages of functional languages can be noted the following the most important, namely**:**
–*programs on functional languagess as a rule are much shorter and simpler than their analogs on imperative languages****;***
–*almost all modern functional languages are strictly typified ensuring the safety of programs****;****strict typification allows to generate more effective code****;***
–*in a functional language the functions can be transferred as an argument to other functions or are returned as result of their call****;***
–*in the pure functional languages (which aren****'****t allowing by****–****effects for functions) there is no an operator of assigning, objects of such language can****'****t be modified and deleted, it is only possible to create new objects by decomposition and synthesis of the existing ones. In pure functional languages all functions are free from by-effects.* Meanwhile, functional

languages can imitate the certain useful imperative properties. Not all functional languages are pure forasmuch in many cases the admissibility of by–effects allows to essentially simplify programming. However today the most developed functional languages are as a rule pure. With many interesting enough questions concerning a subject of functional programming, the reader can familiarize oneself, for example, in [74]. While with quite interesting critical remarks on functional languages and possible ways of their elimination it is possible to familiarize oneself in [28-33,75].

A series of concepts and paradigms are specific for *functional* programming and are absent in*imperative* programming. Meanwhile, many programming languages, as a rule, are based on several paradigms of programming, thus imperative programming languages can quite use and concepts of*functional* programming. In particular, as an important enough concept are so–called pure functions, whose result of performance depends only on their factual arguments. Such functions possess certain useful properties a part of which it is possible to use for optimization of program code and parallelization of calculations. Questions of realization of certain properties of pure functions in the environment of imperative***Maple***–language have been considered in [28-33]. In principle, there are no special difficulties for programming in the functional style in languages that aren**’**t the functional. The***Math***–language professing the mixed paradigm of functional and procedural programming supports functional programming, then the***Maple***–language professing the concept of especially procedural programming at the same time only allows certain elements of functional programming.

However, first of all a few words about the system functions, i.e. functions belonging properly to the ***Math***–language and its environment. Generally speaking, to call these system tools by functions is not entirely correct since realization of many of them is based on the procedural organization, but we stopped on such terminology inherent actually to the system. So, the**Mathematica** system has very large number of the built-in functions, at the same time it provides simple enough mechanisms for definition of the user functions. In the simplest case a certain function***F*** with several formal arguments***x, y, z, …*** has the following very simple format, namely**:**

F[*x*_,*y*_,*z*_,…] {:=|=}***an expression dependent on variables x, y, z,… as a rule***

So, ***F*****[x_]:=x^3+80** defines the function***F(x)=x^3+80*** in standard*mathematical* notation**;** the call of such function on concrete actual argument is defined as ***F*****[*x*],** in particular, as illustrates the following simple example, namely**:** In[2442]**:= F[x_] := x^3 + 90; F[500]**
Out[2442]= 125000090

For receiving of definition of an arbitrary function *(and not only functions,but an arbitrary definition on the basis of operator of postponed**“:=”**or immediate**“=”** assignments)***,** excepting the built–in functions, serve the built–in**Definition** function along with our means considered in the previous section, allowing to receive the optimized definitions in the above sense of as procedures and functions. We will consider briefly elements of functional programming in the**Mathematica** in whose basis the concept of the***pure function*** lays. So, the*pure functions*– one of the basic concepts of*functional* programming that is a component of all programming system in the**Mathematica** in general. Further, the questions relating to this component will be considered in more detail, here we will define only the basic concepts. In the**Mathematica** pure functions are defined as follows.

A pure function in the environment of the **Mathematica** has the following three formats of coding, namely**:**

Function[*x, Function body*] –*a pure function with one formal argument**x;***

Function[{***x1,x2, …,xp***}**,*Function body*] –***a pure function with formal arguments**x1, x2, …,xp;***
(Function body)*& –a pure function with formal arguments**#,#1,#2,…, #n.***

At that, at using of the third format that is often called as short form of pure function for its identification the*ampersand (**&**)* serves whose absence causes either erroneous situations or incorrect results at impossibility to identify the demanded pure function. The reader familiar with formal logic or the***Lisp*** programming language can simply identify pure functions with unnamed functions or**λ–**expressions. Moreover, the pure functions are rather close to mathematical concept of operators. In definition of a*pure* function so**–**called substitutes*(#)* of variables are used, namely**:**

*# –the first variable of a pure function**;***
*#**n** – **n**–th variable of a pure function**;***
*## –sequence of all variables of a pure function**;***
*##**n** –sequence of variables of a pure function starting with**n**–th variable**.***

At application of *pure functions*, unlike traditional functions and procedures, there is no need to designate their names, allowing to code their definitions directly in points of their call that is caused by that the results of the calls of pure functions depend only on values of the actual arguments received by them. So, selection from a list***W*** of the elements meeting certain conditions and elementwise application of a function to elements of a list can be carried out by constructions of the format**Select[*W, Test*[#] &]** and**Map[F[#] &,*W*]** respectively as illustrates the following simple example, namely**:**

```
In[2341] := Select [{a, 72, 75, 42, g,67,Art, Kr, 2015, s,47,500}, IntegerQ[#] &]
Out[2341]= {72, 75, 42, 67, 2015, 47, 500}
In[2342]:= Map[(#^2 + #) &, {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12}]
Out[2342]= {2, 6, 12, 20, 30, 42, 56, 72, 90, 110, 132, 156}
```

At using of the short form of a pure function it is necessary to be careful at its coding because the ampersand has quite low priority. For example, the expression***#1+ #2– #3+ #2&*** without*parentheses* is correct while, generally speaking, they are obligatory, in particular, at using of a*pure* function as the right part of a transformation rule as illustrates very simple example**:** In[2392]**:=** {**a /. a–> #1 + #2 + #3 &, a /. a–> (#1 + #2 + #3 &)**}
Out[2392]= {a /. a–> #1+ #2+ #3**&,** #1+ #2+ #3**&**}
In[2393]**:=** {**Replace[a, a–> #1*#2*#3 &], a /. a–> (#1*#2*#3 &)**}

Replace**::**reps**:** {a–> #1#2#3**&**} is neither a list of replacement rules nor a valid dispatch table, and so cannot be used for replacing. >> Out[2393]= {Replace[a**,** a–> #1#2#3**&**]**,** #1#2#3**&**}

In combination with a number of functions, in particular, **Map, Select** and some others the using of pure functions is rather convenient, therefore the question of converting from traditional functions into pure functions seems a quite topical**;** for its decision various approaches, including creation of the program converters can be used. We used pure functions a rather widely at programming of a number of problems of different types of the applied and the system character [28-33,48].

The following procedure provides converting of a function determined by the format**G[*x*_, *y*_, …]: =*W(x, y, …)*** into pure function of any admissible format, namely**:** the call**FuncToPure[*x*]** returns the pure function that is an analog of a function***x*** of the third format, whereas the call**FuncToPure[*x*, *p*]** where***p* –** any expression, returns pure function of the*two* first formats. The fragment below represents source code of the**FuncToPure** procedure along with typical examples of its usage.

In[2822] **:= FuncToPure[x_ /; QFunction[ToString[x]], y___] := Module[{d, t, a = HeadPF[x], b = Map[ToString, Args[x]], c = Definition2[x][[1]], k = 1, h, g = {}, p}, d = Map[First, Mapp[StringSplit, b, "_"]]; p = StringTrim[c, a <> " := "]; h = "Hold[" <> p <> "]"; {t, h}= {Length[b], ToExpression[h]}; While[k <= t, AppendTo[g, d[[k]] <> "–> #" <> ToString[k]]; k++]; h = ToString1[ReplaceAll[h, ToExpression[g]]]; g = StringTake[h, {6,–2}]; ToExpression[If[{y}!= {}, "Function[" <> If[Length[b] == 1, StringTake[ToString[d], {2,–2}], ToString[d]] <> ", " <> p <> "]", g <> " &"]]]**

In[2823] **:= G[x_Integer, y_Integer, z_Real] := z*(x + y) + Sin[x*y*z]; FuncToPure[G]**
Out[2823]=**#3** (#1+ #2) + Sin[#1#2#3]**&**
In[2824]**:= G[x_Integer, y_Integer, z_Real] := z*(x + y) + Sin[x*y*z]; FuncToPure[G, 80]**
Out[2824]= Function[{x, y, z}, z*(x+ y)+ Sin[x*y*z]]
In[2825]**:= V[x_ /; IntegerQ[x]] := If[PrimeQ[x], True, False];
Select[{47, 72, 25, 18, 480, 13, 7, 41, 561, 2, 123, 322, 17, 23}, FuncToPure[V]]**
Out[2825]= {47, 13, 7, 41, 2, 17, 23}
In[2826]**:= {S[x_] := x^2 + 23*x + 16; FuncToPure[S, 47], FuncToPure[S][80]}**
Out[2826]= {Function[x, x^2+ 23*x+ 16], 8256}

However, at using of the **FuncToPure** procedure for converting of a certain traditional function into pure function it must be kept in mind a number of the essential enough moments. First, the resultant pure function won't have attributes, options and initial values of arguments along with logic tests for admissibility of the actual arguments. Secondly, a converting automatically doesn't do the resultant function as a pure function if a original traditional function such wasn't, i.e. result of the procedure call should depend only on the obtained actual arguments. A number of useful means of operating with pure functions will be considered in the book slightly below.
On the other hand, the following procedure in a certain measure is inverse to the previous procedure, its call**PureToFunc[*x*, *y*]** where***x* –** definition of a pure function, and***y* –** an unevaluated symbol**–** returns*Null***,** i.e. nothing, providing converting of definition of a pure function***x*** into the evaluated definition of equivalent function with a name***y*.** In addition, on inadmissible actual arguments the procedure call is returned unevaluated. The fragment below represents source code of the procedure with an example of its usage.

In[2525]**:= PureToFunc[x_ /; PureFuncQ[x], y_ /; ! HowAct[y]] :=**

Module[{a = Map[ToString, OP[x]], b, c, d, k = 1}, b = Select[a, StringTake[#, {1, 1}] == "#" &]; d = Map[StringReplace[#, "#"–> "x"] &, b]; c = ToString[y] <> "["; For[k, k <= Length[b], k++, c = c <> d[[k]] <> "_,"]; c = StringTake[c, {1,–2}] <> "]

:= "; ToExpression[StringTake[c <> StringReplace[ToString1[x], GenRules[b, d]], {1,–4}]]]

In[2526] **:= PureToFunc[#4*(#1 + #2)/(#3–#4) + #1*#4 &, Gs]** In[2527]**:= Definition[Gs]**
Out[2527]= Gs[x1_, x2_, x3_, x4_]:= (x4 (x1+ x2))/(x3– x4)+ x1 x4

Unlike **FuncToPure** the call**ModToPureFunc[*x*]** provides the converting of a module or block***x*** into pure function under following conditions**:*(1)*** the module/block***x*** can**'**t have local variables or all local variables should have initial values**;*(2)*** the module/block***x*** can**'**t have active global variables, i.e. variables for which in an arbitrary object***x*** assignments are done**;*(3)*** formal arguments of the returned function don**'**t save tests for their admissibility**; *(4)*** the returned function inherits attributes and options of an object***x*.** The fragment below represents procedure code with examples of its application.

In[2428] **:= ModToPureFunc[x_ /; QBlockMod[x]] := Module[{a, c, d, p, j, t, Atr = Attributes[x], O = Options[x], n = "$$$" <> ToString[x], b = Flatten[{PureDefinition[x]}][[1]], k = 1, q = {}}, ToExpression["$$$" <> b]; c = LocalsGlobals1[Symbol[n]]; a = Args[Symbol[n], 80]; d = StringReplace[PureDefinition[n], HeadPF[n] <> " := "–> "", 1]; ToExpression["ClearAll[" <> n <> "]"]; If[c[[3]] != {}, Return[{$Failed, "Globals", c[[3]]}]]; c = Map[{#, ToString[Unique[#]]}&, Join[a, c[[1]]]]; While[k <= Length[c], p = c[[k]]; d = StringReplaceS[d, p[[1]], p[[2]]]; k++]; d = ToString[ToExpression[d]]; t = Map[ToString, UnDefVars[ToExpression[d]]];**

t = Map[StringTake[#, {1, If[StringFreeQ[#, "$"],–1, Flatten[StringPosition[#, "$"]][[1]]–1]}] &, t]; k = 1; While[k <= Length[t], j = 1; While[j <= Length[c], If[t[[k]] == c[[j]][[2]], AppendTo[q, c[[j]][[1]]]]; j++]; k++]; k = 1; While[k <= Length[c], p = c[[k]]; d = StringReplaceS[d, p[[2]], p[[1]]]; k++]; If[p = MinusList[q, a]; p != {}, {$Failed, "Locals", p}, ToExpression["ClearAll[" <> n <> "]"]; n = "$$$" <> ToString[x]; ToExpression[n <> " := Function[" <> ToString[a] <> ", " <> d <> "]"]; If[Atr != {}, ToExpression["SetAttributes[" <> n <> "," <>

ToString[Atr] <> "]"]]; If[O != {}, ToExpression["SetOptions[" <> n <> "," <> ToString[O] <> "]"]]; n]] In[2429]:= **B[x_, y_] := Block[{a, b = 80, c, d}, (a + b + c)*(x + y + d)]; B1[x_, y_] := Block[{a = 480, b = 80, c = 72}, (a + b + c)*(x + y)];**

SetAttributes[B1, {Protected, Listable}]; B2[x_, y_] := Block[{a = 480, b = 80}, h = (a + b)*(x + y); t = 42]; B3[x_, y_] := Block[{a = 480, b, c}, h = (a + b + c)*(x + y); g = 67]; B4[x_, y_] := Block[{a = 480, b = 80}, h = (a + b)*(x + y); t = z]; B5[x_, y_] := Module[{a = 480, b, c, d = 80}, (a + b)*(c + d)]

In[2430] **:= ModToPureFunc[B]**
Out[2430]= {$Failed, "Locals", {"a", "c", "d"}}
In[2431]**:= ModToPureFunc[B1]**
Out[2431]= "$$$B1"
In[2432]**:= Definition["$$$B1"]**
Out[2432]= Attributes[$$$B1]= {Listable, Protected}

$$$B1**:=** Function[{x, y}, 632*****(x+ y)]

In[2433] := **ModToPureFunc[B2]**
Out[2433]= {$Failed, "Globals", {"h", "t"}}
In[2434]:= **ModToPureFunc[B3]**
Out[2434]= {$Failed, "Globals", {"h", "g"}}
In[2435]:= **ModToPureFunc[B4]**
Out[2435]= {$Failed, "Globals", {"h", "t"}}
In[2436]:= **ModToPureFunc[B5]**
Out[2436]= {$Failed, "Locals", {"b", "c"}}
In[2437]:= **A[m_, n_, p_ /; IntegerQ[p], h_ /; PrimeQ[h]] :=**

Module[{a = 42.80}, h*(m+n+p)/a] In[2438]:= **ModToPureFunc[A]**
Out[2438]= "$$$A"
In[2439]:= **Definition["$$$A"]**
Out[2439]= **$$$A:=** Function[{m, n, p, h}, 0.0233645*h*(m+ n+ p)] In[2440]:= **M[x_, y_ /; StringQ[y]] := Module[{a, b = 80, c = 6, d}, a*x + b*y]** In[2441]:= **SetAttributes[M, Protected]; ModToPureFunc[M]** Out[2441]= {$Failed, "Locals", {"a", "d"}}
In[2442]:= **G[x_] := Module[{a = 90, b = 500}, a + b]; ModToPureFunc[G];**

Definition[$$$G]
Out[2442]= **$$$G:=** Function[{x}, 590]

A successful call **ModToPureFunc[*x*]** returns the name of the resultant pure function in the form**ToString [Unique[*x*]],** otherwise procedure call returns the nested list of the format {**$*Failed*,** {*"Locals"|"Globals"*}, {*list of variables in string format*}} whose the first element**$**Failed determines inadmissibility of converting,*second* element**–** the type of the variables that were as a reason of it while the third element– the list of variables of this type in string format. At that, the name of a block or module should be as the actual argument***x*,** otherwise the procedure call is returned unevaluated. Along with standard means the procedure in very essential degree uses our procedures**HeadPF, Args,LocalsGlobals1, ClearAllMinusList, PureDefinition, StringReplaceS, QBlockMod, UnDefVars** that are considered in the present book and in considered in the present book and in 33], that allowed to considerably simplify programming of this procedure. These means and at programming some other appendices are rather useful. In general, it must be kept in mind that the mechanism of the*pure* functions composes a core of the paradigm of*functional programming* in**Mathematica.**

6.3. Means of testing of procedures and functions in the *Mathematica*software

Having defined procedures of two types *(**Module**and**Block**)* and functions, including***pure*** functions, at the same time we have no standard means for identification of objects of the given types. In this regard we created a series of means that allow to identify objects of the specified types. In the present section non–standard means for testing of procedural and functional objects are considered. We will note that the**Mathematica–** a rather closed system in contradistinction, for example, to its main competitor**–** the**Maple** system in which perusal of source codes of its software that are located both in the main and in auxiliary libraries is admissible. While the**Mathematica** system has no similar

opportunity. In this connexion the software presented below concerns only to the user functions and procedures loaded into the current session from a package***(m**–or the**mx**–file)***,* or a document*(**nb**–file**;**also may contain a package)* and activated in it.
It is known that for providing a modularity of the software the procedures are rather widely used that in the conditions of the**Mathematica** system the modular and block constructions provide. Both a module***(Module),*** and a block***(Block)*** provide the closed domain of variables which is supported via the mechanism of local variables. Procedures on the basis of both modular, and block structure provide, in general, a rather satisfactory mechanism of the modularity. Above we attributed the modular and block objects to the procedure type, but here not everything so unambiguously and that is why. In procedural programming a procedure represents some kind of so–called **"***black box***"** whose contents is hidden from the external software with which it interacts only through arguments and global variables*(if they are used by a procedure body)*. Whereas action domain of the local variables is limited by a procedure body only, without crossing with the variables of the same name outside of procedure. Meanwhile, between procedures of*modular* and*block* types exists a rather essential distinction which is based on mechanisms of *local* variables that are used by both types of procedures. In brief the essence of such distinction consists in the following.
Traditional programming languages at work with variables use mechanism **"***lexical scope***",** which is similar to the modular mechanism in**Mathematica,** while the modular mechanism is similar to**"***dynamic scope***"** that is used, for

example, in the symbolic languages like ***Lisp.*** So, if lexical scope considers the local variables connected with a module, dynamic scope considers the local variables connected only with a concrete segment of history of a block execution. In books [28-33] the question of preference of procedures on the basis of modular structure, than on the basis of block structure is considered in details. Meanwhile, the block procedures are often convenient in case of organization of various interactive calculations. Thus, generally, supposing existence of procedures of the above two types***(modular**and**block)*** in the **Mathematica** software**,** for ensuring reliability it is recommended to use the procedures of***Module*** type. Distinctions of procedures on both basics can be illustrated with the following typical examples, namely**:**

```
In[2254] := B[x_] := Block[{a, b, c, d}, x*(a + b + c + d)]
In[2255]:= {a, b, c, d}= {42, 47, 67, 6}
Out[2255]= {42, 47, 67, 6}
In[2256]:= {B[100], a, b, c, d}
Out[2256]= {16200, 42, 47, 67, 6}
In[2257]:= B[x_] := Block[{a = 80, b = 480, c, d}, x*(a + b + c + d)] In[2258]:=
{B[100], a, b, c, d}
Out[2258]= {63300, 42, 47, 67, 6}
In[2259]:= M[x_] := Module[{a = 80, b = 480, c, d}, x*(a + b + c + d)] In[2260]:=
{M[100], a, b, c, d}
Out[2260]= {100 (560+ c$75395+ d$75395), 42, 47, 67, 6}
In[2261]:= {a, b, c, d}= {42, 47, 18, 25};
In[2262]:= B2[x_] := Block[{a, b, c, d}, {a,b,c,d}= {72,67,80,480}; Plus[a,b,c,d]]
In[2263]:= {B2[100], {a, b, c, d}}
```

```
Out[2263]= {699, {42, 47, 18, 25}}
```

From the presented fragment follows, if local variables of a modular object aren't crossed with domain of values of the variables of the same name that are external in relation to it, the absolutely other picture takes place in a case with local variables of a block object, namely: if initial values or values are ascribed to*all* local variables of such object in its body, they save effect in the object body; those variables of object to which such values weren't ascribed accept values of the variables of the same name that are external in relation to the block object. So, at fulfillment of the listed conditions the*modular* and

block objects relative to local variables*(and in general as procedural objects)* can quite be considered as equivalent. Naturally, told remains in force also for block objects with empty lists of local variables. Specified reasons have been laid to the basis of an algorithm programmed by the**RealProcQ** procedure represented by the following fragment.

```
In[2347]:= RealProcQ[x_] := Module[{a, b = " = ", c, d, p},

If[! ProcQ[x], False, If[ModuleQ[x], True, a = Locals1[x]; c = PureDefinition[x]; d = Map[#[[1]]–1 &, StringPosition[c, b]]; p = Map[ExprOfStr[c, #,–1, {" ", "{", "["}] &, d]; p = DeleteDuplicates[Flatten[Map[StrToList, p]]]; If[p == a, True, False]]]]

In[2348] := B1[x_] := Block[{a = 80, b = 480, c = 72, d = 42}, x*(a + b + c + d)]; RealProcQ[B1]
Out[2348]= True
In[2349]:= M2[x_] := Block[{a = 80, b = 480, c, d}, {c, d}= {42, 47};x*a*b*c*d]; RealProcQ[M2]
Out[2349]= True
In[2350]:= M3[x_] := Block[{a = 80, b = 48, c, h}, h = 72; x*h]; RealProcQ[M3]
Out[2350]= False
```

Experience of usage of the **RealProcQ** procedure confirmed its efficiency at testing objects of the type*"Block"* that are considered as real procedures. At that, we will understand an object of type {*Module*,*Block*} as a*real* procedure which in the**Mathematica** software is functionally equivalent to a***Module***, i.e. is some procedure in its classical understanding. The call**RealProcQ[*x*]** returns*True* if the symbol***x*** defines a*Module* or a*Block* which is equivalent to a*Module*, and*False* otherwise. At that, it is supposed that a certain block is equivalent to a module if all its local variables have initial values or some local variables have initial values while others obtain values by the operator **"="** in the block body. The procedure along with the standard means uses as well procedures**ProcQ, Locals1, ModuleQ, ExprOfStr, StrToList** which are considered in the present book and in [33]. From all our means solving the problem of testing of the*procedural* objects, the above**RealProcQ** procedure with the greatest possible reliability identifies the procedure in its classical understanding; in addition, the procedure can be of type {***Module*, *Block***}. In some cases in addition to the above means of testing of the***Math***-objects a rather useful and quite simple procedure can be used whose call**BlockQ[*x*]** returns*True* if the symbol***x*** defines a block object, and*False* otherwise. The following fragment represents source code of the**BlockQ** procedure along with the most typical examples of its usage.

```
In[2377] := BlockQ[x_] := Module[{b, a = If[SymbolQ[x],
```

Flatten[{PureDefinition[x]}][[1]], $Failed]}, If[MemberQ[{$Failed, "System"}, a], False,

b = Mapp[StringJoin, {" := ", " = "}, "Block[{"]; If[SuffPref[a, Map3[StringJoin, HeadPF[x], b], 1], True, False]]] In[2378]:= **Sv[x_] := Module[{}, y := 72; z := 67; {y, z}];**
Agn[x_] := Block[{a = 80}, a*x]; Kr[x_] := Block[{y = a, h = b},

(y^2 + h^2)*x]; Art[x_] := Module[{a = 72}, x*a] In[2379]:= **Map[BlockQ, {Sv, Kr, Agn, Art, a + b, 90}]**
Out[2379]= {False, True, True, False, False, False}

In[2380]:= **BlockQ1[x_Symbol] := If[TestBFM[x] === "Block", True, False]**
In[2381]:= **Map[BlockQ1, {Sv, Kr, Agn, Art, 80}]**
Out[2381]= {False, True, True, False, BlockQ1[80]}
In[2382]:= **ModuleQ1[*x_Symbol*]:= If[*TestBFM*[*x*]===*"Module"*,*True*,*False*]**
In[2383]:= **Map[ModuleQ1, {Sv, Kr, Agn, Art, 90}]**
Out[2383]= {True, False, False, True, ModuleQ1[90]}

In[2384] **:= ModuleQ2[x_] := Module[{b, a = If[SymbolQ[x], Flatten[{PureDefinition[x]}][[1]], $Failed]}, If[MemberQ[{$Failed, "System"}, a], False,**

b = Mapp[StringJoin, {" := ", " = "}, "Module[{"]; If[SuffPref[a, Map3[StringJoin, HeadPF[x], b], 1], True, False]]]

The above fragment is completed by an example with the simple **BlockQ1** function which is functionally equivalent to the previous**BlockQ** procedure and is based on our**TestBFM** procedure**;** this fragment represents also not less simple**ModuleQ1** function whose call**ModuleQ1[*x*]** returns*True* if the symbol***x*** defines a modular structure, and*False* otherwise. The result of the procedure call**ModuleQ2[*x*]** is analogous to the call**ModuleQ1[*x*].** We will note, the previous means of testing of objects of type {*Module*,*Block*, *Function*} support only single objects, but not objects of the same name, i.e. for each such object in the current session of**Mathematica** system the only definition should be activated. Therefore the means of testing of objects in independence from number of the definitions standing behind them are of special interest. Such problem is solved by the following**FuncBlockModQ** procedure, whose result of a call**FuncBlockModQ[*x*, *y*]** returns*True*, if***x*** **–** the symbol defines an object of type {*Function*,*Module*,*Block*}**;** at that, in the presence for the symbol***x*** of several definitions the*True* is returned only if all its definitions generate an object of the same type. Whereas through the second argument***y*** ***–an undefinite variable–*** an object type in the context of {*"Function"*, *"Block"*, *"Module"*} is returned. If symbol***x*** defines an object of the same name whose definitions are associated with subobjects of different types, the procedure call**FuncBlockModQ[*x*,*y*]** returns*False* while thru the ***2nd*** argument***y*** ***"Multiple"*** is returned. The following fragment represents source code of the**FuncBlockModQ** procedure along with the most typical examples of its usage.

In[2654]:= **FuncBlockModQ[x_ /; SymbolQ[x], y_ /; ! HowAct[y]] := Module[{b, c, m, n, a = PureDefinition[x]},**

If[MemberQ[{"System", $Failed}, a], False, a = Flatten[{a}]; b =

Flatten[{HeadPF[x]}]; c = Join[Mapp[StringJoin, b, " := "], Mapp[StringJoin, b, " = "]]; c = GenRules[c, ""]; c = StringReplace[a, c]; {m, n}= Map[Length, {Select[c, SuffPref[#, "Block[{", 1] &], Select[c, SuffPref[#, "Module[{", 1] &]}]; If[Length[a] == m, y = "Block"; True,

If[Length[a] == n, y = "Module"; True,
If[m + n == 0, y = "Function"; True, y = "Multiple"; False]]]]]

In[2655] **:= Sv[x_] := Module[{}, y := 72; z := 667; {y, z}];**
Agn[x_] := Block[{a = 80}, a*x]; B[x_] := Block[{a, b, c, d}, x*(a + b + c + d)]; B[x_, y_] := Block[{}, x + y]; M[x_] := Module[{a, b, c, d}, x*(a + b + c + d)]; M[x_, y_] := Module[{}, x + y]; V[x_] := Module[{a, b, c, d}, x*(a + b + c +d)]; V[x_, y_] := Block[{}, x + y]; F[x_, y_] := x + y; F[x_, y_, z_] := x + y + z

In[2656]**:=** {**FuncBlockModQ[Sv, y], y**}

Out[2656] = {True**,** "Module"}
In[2657]**:=** {**FuncBlockModQ[B, y1], y1**}
Out[2657]= {True**,** "Block"}
In[2658]**:=** {**FuncBlockModQ[M, y2], y2**}
Out[2658]= {True**,** "Module"}
In[2659]**:=** {**FuncBlockModQ[V, y3], y3**}
Out[2659]= {False**,** "Multiple"}
In[2660]**:=** {**FuncBlockModQ[While, y4], y4**}
Out[2660]= {False**,** y4}
In[2661]**:=** {**FuncBlockModQ[F, y4], y4**}
Out[2661]= {True**,** "Function"}

This procedure along with standard means uses also our means **GenRules, HeadPF, HowAct, Mapp, PureDefinition, SuffPref** and**SymbolQ** that are considered in this book and in [32,33]. Below, other means of testing of the objects of type {*"Function","Block"*, *"Module"*} will be presented too, though already the above means allow to considerably solve the given problem.

Insofar as procedures of both types ***(Module, Block)*** along with functions of the user are basic objects of procedural programming in the**Mathematica** then a very important problem of creation of means for testing of belonging of an object to the type {***Module, Block, Function***} exists. The next fragment represents the**TestBFM** procedure that is successfully solving this problem.

In[2620]**:= TestBFM[x_] := Module[{a = Flatten[{PureDefinition[x]}], b, d, h, p, k, j, t = {}},**

If[MemberQ[{$Failed, "System"}, a[[1]]], Return[$Failed], b = Flatten[{HeadPF[x]}]; For[k = 1, k <= Length[a], k++, d = a[[k]]; p = Map[b[[k]] <> # &, {" := ", " = "}]; h = StringReplace[d, {p[[1]]–> "", p[[2]]–> ""}, 1]; If[SuffPref[h, "Module[{", 1], t = AppendTo[t, "Module"], If[SuffPref[h, "Block[{", 1], t = AppendTo[t, "Block"],

If[SuffPref[h, "DynamicModule[{", 1],
t = AppendTo[t, "DynamicModule"], t = AppendTo[t, "Function"]]]]]]; If[Length[t]

== 1, t[[1]], t]]

In[2621] **:= M[x_] := x; M[x_, y_] := Module[**{}**, x + y];**
M[x_, y_, z_] := Block[{}**, x + y + z];**
In[2622]**:= PureDefinition[M]**
Out[2622]= {"M[x_]:= x", "M[x_, y_]:= Module[{}, x+y]",
"M[x_, y_, z_]:= Block[{}, x+y+z]"}
In[2623]**:= TestBFM[M]**
Out[2623]= {"Function", "Module", "Block"}
In[2624]**:= Map[TestBFM,** {**a + b, avz, Sin, SuffPref, For, 2015**}**]**
Out[2624]= {$Failed, $Failed, $Failed, "Module", $Failed", $Failed}

The procedure call **TestBFM[*x*]** returns the type of a functional, modular or block object***x*** in format***"Function", "Module", "DynamicModule", "Block",*** whereas on argument***x*** of other type the procedure call returns**$Failed.** At that, if an argument***x*** defines an object of the same name, the procedure call **TestBFM[*x*]** returns the list of types of the subobjects composing it, having bijection with the list of definitions returned by the call**PureDefinition[*x*].**

At that, the following procedure can appear as an useful enough means of testing of objects, its call**ProcFuncBlQ[*x, y*]** returns*True* if***x*** is a procedure, function or block, otherwise*False* is returned. Moreover, at return of*True*, thru argument***y*** *–an undefinite variable–* a***x***object type is returned {*"Block", "Module", "DynamicModule", "Function", "PureFunction"*}, otherwise the***2nd*** argument remains undefinite. The next fragment represents source code of the procedure along with the most typical examples of its usage.

In[3178]:= **ProcFuncBlQ[x_, y_ /; ! HowAct[y]] :=**

Module[{**a = ToString[HeadPF[x]], b = ToString[y] <> " = ", c = PureDefinition[x]**}**, If[ListQ[c], False, If[SuffPref[a, "HeadPF[", 1], If[SuffPref[a, " &]", 2], y = "PureFunction"; True, False], If[HeadingQ[a],**
If[SuffPref[c, a <> " := Module[{**", 1], y = "Module"; True, If[SuffPref[c, a <> " := Block[**{**", 1], y = "Block"; True, If[SuffPref[c, a <> " := DynamicModule[**{**", 1],**

y = "DynamicModule"; True, y = "Function"; True]]], False]]]] In[3179]:= **Dm[] := DynamicModule[**{**x**}**,** {**Slider[Dynamic[x]], Dynamic[x]**}**]**

In[3180] **:= DPOb[] := Module[**{**a = 80, b = 67, c = 18, d = 25**}**, Plus[a, b, c, d]]**
In[3181]**:= B[x_] := Block[**{**a**}**, a = x]; G := Function[500 + 90*# &];** In[3182]**:= Clear[g, g1, g2, g3, g4, g5];** {**ProcFuncBlQ[Dm, g],**

ProcFuncBlQ[DPOb, g1], ProcFuncBlQ[B, g2], ProcFuncBlQ[G, g3], ProcFuncBlQ[500 + 90*# &, g4], ProcFuncBlQ[500, g5]} Out[3182]= {True, True, True, True, True, False}
In[3183]**:=** {**g, g1, g2, g3, g4, g5**}
Out[3183]= {"DynamicModule", "Module", "Block", "PureFunction", "PureFunction", g5}
In[3184]**:= ClearAll[t]; F[x_] := 500 + 90*x;** {**ProcFuncBlQ[F, t], t**} Out[3184]= {True, "Function"}

It should be noted that this procedure is correctly executed only on objects of the above type provided that they have the single definitions, otherwise returning the*False***.** The procedure along with standard means uses also our means**HeadingQ, HeadPF, HowAct, PureDefinition** and**SuffPref** that are considered in the present book and in our previous books [32,33].

As it was already noted above, in general case between procedures of types **"***Module***"** and**"***Block***"** exist principal enough distinctions which don't allow a priori to consider a block structure as a full procedure. Such distinctions are based on various used mechanisms of local variables as it was visually illustrated with examples slightly above. It is possible to give more complex examples of similar distinctions [30-33]. Therefore the type of a procedure should be chosen rather circumspectly, giving preference to procedures of the type**"***Module***".** Therefore, the**BlockToModule** procedure can be usefull enough, whose call**BlockToModule[*x*]** returns*Null***,** providing converting of a procedure of the type**"***Block***"** into procedure of the type**"***Module***".** The fragment below represents source code of the**BlockToModule** procedure along with typical examples of its usage.

In[2468] **:= BlockToModule[x_Symbol] := Module[{b, c, d, h = {}, k = 1, n, m, a = Definition2[x]}, If[ListQ[a] && a[[1]] == "System" || UnevaluatedQ[Definition2, x], $Failed, b = a[[–1]]; ClearAllAttributes[x]; c = a[[1 ;;–2]]; d = Flatten[{HeadPF[x]}]; For[k, k <= Length[d], k++, {n, m}= {c[[k]], d[[k]]}; If[SuffPref[n, {m <> " := Block[{", m <> " = Block[{"}, 1], AppendTo[h, StringReplace[n, "Block[{"–> "Module[{", 1]], AppendTo[h, n]]]; Map[ToExpression, h]; SetAttributes[x, b]]]**

In[2469] **:= V[x_] := Module[{a, b}, x*(a + b)]; V[x_, y_] := Block[{}, x + y]; V[x__] := {x}**
In[2470]**:= Options[V] = {agn–> 67, asv–> 47};**
SetAttributes[V, {Protected, Listable}]
In[2471]**:= Definition2[V]**
Out[2471]= {"V[x_]:= Module[{a, b}, x*(a+b)]", "V[x_, y_]:= Block[{}, x+y]", "V[x__]:= {x}", "Options[V]= {agn–> 67, asv–> 47}", {Listable, Protected}}
In[2472]**:= BlockToModule[V]**
In[2473]**:= Definition2[V]**
Out[2473]= {"V[x_]:=Module[{a, b}, x*(a+b)]", "V[x_, y_]:=Module[{}, x+y]", "V[x__]:= {x}", "Options[V]= {agn–> 67, asv–> 47}", {Listable, Protected}}
In[2474]**:= G[x_] := Block[{}, x^2]; G[x_, y_] = Block[{}, x * y]; G[x__] := Block[{}, {x}]**
In[2475]**:= Options[G] = {ian–> 80, rans–> 480};**
SetAttributes[G, {Protected, Listable}]
In[2476]**:= Definition2[G]**
Out[2476]= {"G[x_]:=Block[{},x^2]", "G[x_, y_]=x*y", "G[x__]:=Block[{}, {x}]", "Options[G]= {ian–> 80, rans–> 480}", {Listable, Protected}}
In[2477]**:= BlockToModule[G]; Definition2[G]**
Out[2477]= {"G[x_]:=Module[{}, x^2]", "G[x_, y_]=x*y", "G[x__]:=Module[{}, {x}]", "Options[G]= {ian–> 80, rans–> 480}", {Listable, Protected}}

The call **BlockToModule[*x*]** returns*Null***,** i.e. nothing, simultaneously with converting of a procedure***x*** of*block* type into the procedure of*modular* type of the same name with preservation of all attributes and options of a source procedure of block type. Moreover,

several definitions of*modules***,***blocks* or/ and*functions* also can be associated with an object***x*****,** however the procedure call**BlockToModule[*x*]** provides converting only of*block* components of the object***x*** into*modular* structures. The above examples quite visually illustrate the aforesaid.
Due to the mechanism of the global variables used by blocks and modules it is necessary to make certain explanations. In this context it is possible to distinguish two types of global variables**–** ***passive*** and***active*** ones. Passive global variables are characterized by that, they are only used by an object, without changing their values outside of the object. While the assignment of values by means of operators {**":="**,**"="**} for*active* global variables is done in an object body, changing their values and outside of the object. In view of the above the active global variables are of interest at processing of blocks and modules, and procedures in general. A number of our means processing the objects of this type whose definitions contain the active global variables consider the specified circumstance, carrying out processing of objects of the type {*"Module"*,*"Block"*} so that not to change values of*active* global variables used by them outside of their scope. In this relation the procedures**BlockQ, ModuleQ,BlockFuncModQ,BlockModQ** given below are rather indicative.

The call **BlockFuncModQ[*x*]** returns*True***,** if***x*** **–** a symbol defining a typical *(with heading)* function, block or module, and*False* otherwise. While the call **BlockFuncModQ[*x*, *y*]** on condition of the main return of*True* through the ***2nd*** optional argument***y*** **–***an undefinite variable***–** returns type of an object***x*** in the context of {*"Block***"**,*"Function***"**, *"Module"*}. On the other hand, the call **BlockModQ[*x*]** returns*True***,** if***x*** **–** symbol defining a block or module, and *False* otherwise. Whereas the call**BlockModQ[*x*, *y*]** on condition of the main return of*True* through optional argument***y*** **–***an undefinite variable***–** returns type of an object***x*** in the context of {*"Block***"**,*"Module"*}. The fragment below submits source codes of the procedures**BlockModQ** and**BlockFuncModQ** along with the most typical examples of their usage.

In[2612]:= **BlockFuncModQ[x_, y___] := Module[{b, c,
a = Flatten[{PureDefinition[x]}][[1]]},**

**If[MemberQ[{$Failed, "System"}, a], False, b = StringSplit[a, {" := ", " = "}, 2];
If[StringFreeQ[b[[1]], "["], False, c = If[SuffPref[b[[2]], "Module[{", 1], "Module",
If[SuffPref[b[[2]], "Block[{", 1], "Block", "Function"]]; If[{y}!= {}&& ! HowAct[y],
y = c]; True]]]**

In[2613] **:= M[x_, y_] := Module[{a = 80, b = 480}, x*y*a*b];
F[x_] := x; B[_] := Block[{}, x]**
In[2614]:= {**BlockFuncModQ[M, y], y**}

Out[2614] = {True**,** "Module"}
In[2615]:= {**BlockFuncModQ[F, y1], y1**}
Out[2615]= {True**,** "Function"}
In[2616]:= {**BlockFuncModQ[B, y2], y2**}
Out[2616]= {True**,** "Block"}

In[2639]:= **BlockModQ[x_, y___] := Module[{s = FromCharacterCode[6], a =
Flatten[{PureDefinition[x]}][[1]], b, c},**

If[MemberQ[{$Failed, “System”}, a], False, b = StringReplace[a, {” := “–> s, ” = “–> s}, 1]; b = StringTake[b, {Flatten[StringPosition[b, s]][[1]] + 1,–1}]; c = If[SuffPref[b, “Module[{“, 1], “Module”, If[SuffPref[b, “Block[{“, 1], “Block”]]; If[{y}!= {}&& ! HowAct[y], y = c]; If[c === Null, False, True]]]

In[2640] := {**BlockModQ[M, y3], y3**}
Out[2640]= {True, “Module”}
In[2641]:= {**BlockModQ[F, y4], y4**}
Out[2641]= {False, Null}
In[2642]:= {**BlockModQ[B, y5], y5**}
Out[2642]= {True, “Block”}

From the aforesaid follows, at programming of the means that manipulate with objects of the type {*”Block*“, “*Module*”} and which use global variables, it is necessary to consider possibility, what in the course of the call of these means for their global variables the assignments are done what can conflict with values of variables of the same name which have been received in the current session earlier. Naturally, in general, that isn’t so essential for the reason that by a call of such objects, the*global* variables used by them and so will receive values if is being not envisaged the contrary. In order to avoid possible misunderstanding a procedure has to provide saving of values of global variables which have been received by them up to the procedure call with restoration them at exit from the procedure. Simple example illustrates a mechanism of saving of values of a variable***y*** of the current session that is used as global variable of a simple procedure***Kr,*** namely:
In[2495]:= **Kr[x_] := Module[{a = 90, b = y}, y = 500; {a + y + x, y = b}[[1]]]**
In[2496]:= **y = 42;** {**Kr[100], y**}

Out[2496]= {690, 42}

Functions of the **Mathematica** system have a number of interesting means for support of work with dynamic objects. We recall that dynamic module **DynamicModule[**{***x,y, z, …***}**,*W*]** represents an object that supports the same local status for variables***x,y, z, …*** in the course of evaluation of all dynamic objects of a***W*** body. The variables specified in**DynamicModule** by default have values throughout all current session. At that, the dynamic object can act not only directly as an expression, but also, in particular, as coordinate in a graphic primitive, as an object of type*”slider“*, as a setting for an option. Meanwhile, unlike the*standard* module the*dynamic* module directly doesn’t allow to receive its definition by the standard**Definition** function, only our procedures**Definition2** and**PureDefinition** allow to solve this problem as it illustrates the following fragment, namely:

In[2760] **:= Dm[x_, y_ /; PrimeQ[y]] := DynamicModule[**{**a = 90, b = 500**}**, a + b*(x + y)]; Definition[Dm]**
Out[2760]= Dm[x_, y_ /; PrimeQ[y]]:= a$$ + b$$ (x+ y)
In[2761]:= **Definition2[Dm]**
Out[2761]= {”Dm[x_, y_ /; PrimeQ[y]]:= DynamicModule[{a= 90, b= 500}, a+ b*(x+ y)]“, {}}
In[2762]:= **PureDefinition[Dm]**
Out[2762]= “Dm[x_, y_ /; PrimeQ[y]]:= DynamicModule[{a= 90, b= 500}, a+ b*(x+ y)]”

In[2799]:= **ModuleQ[x_Symbol, y___ /; y == Null || SymbolQ[y] && ! HowAct[y]] := Module[{a = PureDefinition[x], b},**

If[ListQ[a] ||a == "System" || a === $Failed, False, b = HeadPF[x]; If[SuffPref[a, b <> " := " <> "Module[{", 1], If[{y}!= {}, y = "Module"]; True,

If[SuffPref[a, b <>" := " <> "DynamicModule[{", 1], If[{y}!= {}, y = "DynamicModule"]; True, False]]]]

In[2800] := {**ModuleQ[Dm, t], t**}
Out[2800]= {True, "DynamicModule"}
In[2801]:= **V[x_] := Module[{a, b}, x*(a + b)];** {**ModuleQ[V, t1], t1**} Out[2801]= {True, "Module"}
In[2802]:= **V[x_, y_] := Block[{}, x + y]; V[x__] := {x};** {**ModuleQ[V, t2], t2**}

Out[2802] = {False, t2}
In[2803]:= {**ModuleQ[Sin, t2], t2**}
Out[2803]= {False, t2}
In[2804]:= {**ModuleQ[500, t2], t2**}
Out[2804]= {ModuleQ[500, t2], t2}

The rather useful **ModuleQ** procedure completes the given fragment whose call**ModuleQ[*x*]** returns*True* if an object***x*,** given by a symbol, is a module, and*False* otherwise**;** while the call**ModuleQ[*x*,*y*]** with the second optional argument***y*** *–an undefinite variable*– through***y*** returns module type***x*** in the context {*"Module"*, *"DynamicModule"*}. At that, the procedure call on a tuple of incorrect actual arguments is returned unevaluated. In other cases the call **ModuleQ[*x*, *y*]** returns the*False***.** The procedure along with standard means uses also our means**HeadPF, HowAct, PureDefinition, SymbolQ, SuffPref** that are considered in this book and in [32,33]. Meanwhile, several essential enough moments concerning the**ModuleQ** procedure should be noted. First of all, the**ModuleQ** procedure is oriented on a modular object***x*** which has single definition, returning*False* on the objects of the same name. Moreover, the procedure algorithm assumes that the definition of a modular object***x*** is based on the operator of*postponed* assignment**":=",** but not on the operator **"="** of the*immediate* assignment because in the latter case the object***x*** will be distinguished by the standard**Definition** function and our testing means as a function. In our opinion, the**ModuleQ** is rather useful in programming of various type of problems and first of all the system character.

For testing of objects onto procedural type we proposed a number of means among which it is possible to note such as**ProcQ, ProcQ1, ProcQ2.** The call **ProcQ[*x*]** provides testing of an object***x*** be as a procedural object {*"Module"*, *"Block"*}**,** returning accordingly*True* or*False;* whereas the**ProcQ1** procedure is a useful enough modification of the**ProcQ** procedure, its call**ProcQ1[*x*, *t*]** returns*True,* if***x*** **–** an object of type***Block*,*Module*** or***DynamicModule*,** and *"Others"* or*False* otherwise**;** at that, the type of object***x*** is returned through the actual argument***t*** *–an undefinite variable.* Source codes of the mentioned procedures, their description along with the most typical examples of their application are presented in our books [30-33] and in***AVZ_Package*** package [48]. A number of receptions used at their creation can be useful enough in practical programming. The above**ProcQ** procedure is quite fast, processes attributes and options, however has certain restrictions, first of all, in case of objects of the same name [33]. The fragment below represents source codes of both procedures along with typical examples of their usage.

In[2492]:= **ProcQ[x_] := Module[{a, atr = Quiet[Attributes[x]], b, c, d, h}, If[! SymbolQ[x], False, If[SystemQ[x], False,**

If[UnevaluatedQ[Definition2, x], False, If[ListQ[atr] && atr != {}, ClearAllAttributes[x]]; a = Quiet[SubsDel[ToString[InputForm[Definition[x]]], "`" <> ToString[x] <> "`", {"[", ",", " "},–1]]; Quiet[b = StringTake[a, {1, First[First[StringPosition[a, {" := Block[{", " :=Block[{"}]–1]]}]; c = StringTake[a, {1, First[

First[StringPosition[a, {" := Module[{"," :=Module[{"}]–1]]}]; d = StringTake[a, {1, First[First[StringPosition[a, {" := DynamicModule[{", " :=DynamicModule[{"}]–1]]}]]; If[b === ToString[HeadPF[x]], SetAttributes[x, atr]]; True, If[c === ToString[HeadPF[x]], SetAttributes[x, atr]]; True, If[d === ToString[HeadPF[x]], SetAttributes[x, atr]]; True, SetAttributes[x, atr]]; False]]]]]]

In[2493]:= **Map[ProcQ, {Sin, a + b, ProcQ1, ProcQ, 73, UnevaluatedQ}]** Out[2493]= {False, False, True, True, False, True}

In[2620] := **ProcQ1[x_, y___ /; y == Null || SymbolQ[y] && ! HowAct[y]] := Module[{a = Quiet[Check[Flatten[{PureDefinition[x]}], $Failed]], b = StringLength[ToString[x]], c, g = ToString[Unique["agn"]], h = {}, p = $$$72, k = 1, t = {}}, If[SubsetQ[{$Failed, "System"}, a], False, For[k, k <= Length[a], k++, Clear[$$$72]; ToExpression[g <> StringTake[a[[k]], {b + 1,–1}]]; AppendTo[h, c = ProcQ[g]]; BlockFuncModQ[g, $$$72]; AppendTo[t, If[c && $$$72 == "Function", "DynamicModule", $$$72]]; Clear[g]; g = ToString[Unique["agn"]]]; $$$72 = p; Clear["$$$72", g]; If[{y}!= {}, y = {h, t}, Null]; If[DeleteDuplicates[h] == {True}, True, False]]]** In[2621]:= **V[x_] := Module[{a, b}, x*(a + b)]; V[x_, y_] := Block[{}, x + y]; V[x__] := {x}; {ProcQ1[V, y], y}**

Out[2621]= {False, {{True, True, False}, {"Module", "Block", "Function"}}}
In[2622]:= **G[x_] := Module[{a = 73}, a*x^2]; G[x_, y_] := Module[{}, x*y]; G[x__] := Module[{a = 90, b = 500}, Length[{x}]+ a*b]; {ProcQ1[G, u], u}** Out[2622]= {True, {{True, True, True}, {"Module", "Module", "Module"}}} In[2686]:= **ProcBMQ[x_ /; BlockModQ[x], y___] :=**
Module[{a, b, c = " = ", d, p}, If[! SingleDefQ[x],

"Object <" <> ToString[x] <> "> has multiple definitions", If[ModuleQ[x], True, {a, b}= {PureDefinition[x], Locals1[x]}; d = Map[#[[1]]–1 &, StringPosition[a, c]]; p = Map[ExprOfStr[a, #,–1, {" ", "{", "["}] &, d]; p = DeleteDuplicates[Flatten[Map[StrToList, p]]]; If[{y}!= {}, y = MinusList[b, p], Null]; If[p == b, True, False]]]]

In[2687] := **P[x_] := Block[{a = 90, b = 500, c, d, h, g}, h = (a + b)*x; h^2]; {ProcBMQ[P, q], q}**
Out[2687]= {False, {"c", "d", "g"}}
In[2688]:= **T[x_] := Block[{a = 6, b = 8, c, d, h, g}, {c, d, h, g}= {1, 2, 3, 4}]; {ProcBMQ[T, v], v}**

```
Out[2688]= {True, {}}
In[2689]:= G[x_] := Block[{a, b}, x]; G[x_, y_] := Block[{a, b}, x + y]; ProcBMQ[G]
Out[2689]= "Object<G> has multiple definitions"

In[2690]:= SingleDefQ[x_] := If[ListQ[PureDefinition[x]] || MemberQ[{$Failed,
"System"}, PureDefinition[x]], False, True]

In[2691] := G[x_] := Block[{}, x]; G[x_, y_] := Block[{a}, x*y]; SingleDefQ[G]
Out[2691]= False
In[2692]:= a[x_] := x; a[x_, y_] := x/y; Map[SingleDefQ, {73, c/b, If, ProcQ, a}]
Out[2692]= {False, False, False, True, False}
```

In this context we created the **ProcQ1** procedure that generalizes the**ProcQ** procedure, first of all, in case of the objects of the same name. The previous fragment represents source code of the**ProcQ1** procedure with examples of its most typical application. The call**ProcQ1[*x*]** returns*True* if the symbol***x*** defines a procedural object of the type {*Block*,*Module*,*DynamicModule*} with unique definition along with an object consisting of their any combinations with different headings*(the objects of the same name)*. Moreover, in case of a separate object or an object***x*** of the same name*True* is returned only when all its components is procedural objects in the sense stated above, i.e. they have a type {*Block*,*DynamicModule*,*Module*}. Meanwhile, the procedure call **ProcQ1[*x*, *y*]** with the***2nd*** optional argument***y*** *–an undefinite variable–* thru it returns simple or the nested list of the following format, namely:

{{***a1, a2, a3, a4, …, ap***}, {***b1, b2, b3, b4, …, bp***}}

where ***aj***∈{*True*,*False*} whereas***bj***∈{*"Block"*, *"DynamicModule"*, *"Function"*, *"Module"*}; at that, between elements of the above sublists exists one-to-one correspondence while pairs {***aj***,***bj***}*(j=1..p)* correspond to subobjects of the object***x*** according to their order as a result of the call**Definition[*x*].**

The **ProcQ1** procedure is rather widely used and is useful enough in many appendices, it differs from the previous**ProcQ** procedure in the following context, namely:***(1)****quite successfully processes the objects of the same name*, ***(2)*** *defines procedurality in case of the objects of the same name*,*whose subobjects are blocks or functions*. The procedure along with standard means significantly uses as well our means such as**HowAct, PureDefinition, SymbolQ, ProcQ, BlockFuncModQ** that are considered in the present book and in [28-33]. At last, the above fragment is completed by the**ProcBMQ** procedure whose call**ProcBMQ[*x*]** with one argument returns*True*, if a block or a module***x*** – a*real* procedure in the above context, and*False* otherwise; the procedure call **ProcBMQ[*x*,*y*]** with the second optional argument***y*** *–an undefinite variable–* returns thru it the list of local variables of the block***x*** in string format which have no initial values or for which in a body of the block***x*** the assignments of values weren't made. We will note, the**ProcBMQ** procedure is oriented only on one-defined objects whose definitions are unique while the message "*Object <****x****> has multiple definitions*" is returned on objects***x*** of the same name. The procedure along with standard means uses also our means**ExprOfStr, BlockModQ, ModuleQ, PureDefinition, Locals1, SingleDefQ, MinusList, StrToList** that are considered in this book and in [30-33]. In particular, the procedure significantly uses rather simple and very useful function, whose call**SingleDefQ[*x*]** returns*True* if the actual argument*x* defines a name of a procedure, a block or a function having single definition**;** in other cases the function call returns*False***.** The above fragment contains source code of the **SingleDefQ** function with the most typical examples of its application. In addition to our means testing procedural objects, we will note the simple procedure, whose call**UprocQ[*x*]** returns*False* if an object***x*** isn**'**t procedure or is object of the same name, and the***2*–**element list otherwise**;** in this case its*first* element is*True* while the*second***–** a type {*"DynamicModule"***|***"Block"***|** *"Module"*} of the object***x*.** On functions the***2***-element list of the format {*False, "Function"*} is returned. On inadmissible factual argument***x*** the procedure call is returned unevaluated. The following fragment represents source code of the**UprocQ** procedure along with typical examples of its application.

In[2515]**:= UprocQ[x_ /; SymbolQ[x]] := Module[**{**a = Unique["agn"], b**}**,**
If[SingleDefQ[x], b = ProcQ1[x, a]; {**b, a[[2]][[1]]**}**, False]]**

In[2516] **:= a[x_] :=** x^3**; Dm[] := DynamicModule[**{x}**,** {**Slider[Dynamic[x]], Dynamic[x]**}**]; P[x_] := Block[**{**a = 90, b = 500, h**}**, h = a*b*x; h^2]**
In[2517]**:= Map[UprocQ,** {**ProcQ, P, Dm, 73, a**}**]**
Out[2517]= {{True**, "**Module"}**,** {True**, "**Block"}**,** {True**, "**DynamicModule"}**,** UprocQ[73]**,** {False**, "**Function"}}

Having considered the main means of testing of procedural objects that are absent among standard means of the**Mathematica** system it is reasonable to consider the means similar to them for testing of the*functional* objects where under functional means we will understand objects whose definitions have the following format, namely**:**

F* [*x*_ /;*Test*$_x$,*y*_ /;*Test*$_y$,*z*_ /;*Test*$_z$,…] :=*W(x, y, z, …) or pure functions of one of the following formats, namely**:**

Function[*Body*]*or short form***Body&***(formal arguments***#(#1), #2, #3,***etc.)* **Function[*x*, *Body*]–***a pure function with single formal argument***x*** **Function[**{***x1, x2, …***}**,*Body*]–***a pure function with formal arguments* {***x1, x2, …***}

We will give some simple examples onto these types of functions, namely**:**

In[2325] **:= y := Function[**{**x, y**}**, x + y]; y1 = Function[**{**x, y**}**, x + y]; z := #1 + #2 &; z1 = #1 + #2 &; F[x_, y_] := x + y**
In[2326]**:=** {**y[80, 480], y1[80, 480], z[80, 480], z1[80, 480], F[80, 480]**}

Out[2326]= {560**,** 560**,** 560**,** 560**,** 560}

On objects of the above functional type the calls of procedures **ProcQ1** and **ProcQ** return*False***,** therefore for testing of functional type and other means considered below are offered. However, first of all, we will consider means testing the system functions, i.e. functions of the***Math*–**language along with its environment. By and large, these system tools are called by***functions*** not entirely correctly, because implementation of many of them is based on the procedural organization, meanwhile, we stopped on the given terminology, inherent actually to the system. And in this regard it is possible to present means of testing of the system functions, besides that, the testing of objects regarding to be standard functions of the**Mathematica** system in a number of important enough problems arises need. In this regard a simple enough function**SysFuncQ** solves the given problem**;** its call**SysFuncQ[*x*]** returns *True* if an object***x*** is a*standard* function of the**Mathematica** system and*False* otherwise**;** whereas simple**SysFuncQ1** function is a functionally equivalent modification of the previous**SysFuncQ** procedure. The following fragment represents source codes of the above means with examples of their usage.

In[2419]**:= SysFuncQ[x_] := If[UnevaluatedQ[Definition2, x], False,**

If[SameQ[Definition2[x][[1]], "System"], True, False]] In[2420]**:= Map[SysFuncQ,** {**Sin, Tan, While, If, Do, ProcQ, 6, Length, a/b**}**]** Out[2420]= {True**,** True**,** True**,** True**,** True**,** False**,** False**,** True**,** False} In[3037]**:= SysFuncQ1[x_] := MemberQ[Names["System`*"], ToString[x]]**

In[3038]**:= Map[SysFuncQ1,** {**Sin, Tan, While, If, Do, ProcQ, 6, Length, a/b**}**]** Out[3038]= {True**,** True**,** True**,** True**,** True**,** False**,** False**,** True**,** False}

We will consider means of testing of the user functional objects, the first of which is the procedure**QFunction** that is the most general means of testing of objects***x*** of the functional type, whose call**QFunction[*x*]** returns*True* on a traditional function***x*** and***x*–**objects, generated by the function**Compile,** and*False* otherwise. At that, the construction of format**J[*x_*, *y_*, …]** {:= | =} ***J(x, y, …)*** is understood as the traditional function. The fragment represents source code of the**QFunction** procedure with examples of its usage. At that, the given procedure along with standard means uses and our means such as **HeadPF,Definition2, SymbolQ, Map3, SuffPref, ToString1** and**ToString3** that are considered in the present book and in [28-33]. In particular, simple **ToString3** function is presented right there and its call**ToString3[*x*]** serves for converting of an expression***x*** in string***InputForm*** format. This function has a number of useful enough appendices.

In[2393]:= **QFunction[x_] := Module[**{**a = Quiet[Definition2[x][[1]]], b = ToString3[HeadPF[x]]**}**,**

If[! SingleDefQ[x], False, If[SameQ[a, x], False, If[SuffPref[Quiet[ToString1[a]], "CompiledFunction[", 1], True, If[SuffPref[b, "HeadPF[", 1], False, b = Map3[StringJoin, b, {**" := ", " = "**}**]; If[MemberQ[**{**SuffPref[StringReplace[a, b –> ""], "Module[", 1], SuffPref[StringReplace[a, b –> ""], "Block[", 1]**}**, True], False, True]]]]]]**

In[2394] **:= V := Compile[**{{**x, _Real**}**,** {**y, _Real**}}**, x/y]; Kr := (#1^2 + #2^4) &; Art := Function[**{**x, y**}**, x*Sin[y]]; GS[x_ /; IntegerQ[x], y_ /; IntegerQ[y]] := Sin[75] + Cos[42]; Sv[x_ /; IntegerQ[x], y_ /; IntegerQ[y]] := x^2 + y^2; S := Compile[**{{**x, _Integer**}**,** {**y, _Real**}}**, (x + y)^3];**

In[2395] **:= Map[QFunction,** {**V, S, Art, Kr, Pi, 42.72, GS, Sv**}**]** Out[2395]= {True**,** True**,** False**,** False**,** False**,** False**,** True**,** True}
In[2396]**:= G[x_Integer, y_Real, z_Real] := x*y^2 + z**
In[2397]**:= Map[QFunction,** {**#1*#2*#3 &, Function[**{**x,y,z**}**, x*y*z],G,ProcQ**}**]**
Out[2397]= {False**,** False**,** True**,** False}

In[2571]**:= ToString3[x_] := StringReplace[ToString1[x], "\"" –> ""]** In[2576]**:= ToString3[#1^2 + #2^4 &]**
Out[2576]= **"**#1^2+ #2^4**&** "

In[2642] **:= QFunction1[x_] := Module[**{**a, c = ToString[Unique["agn"]], b, p, d =** {}**, k = 1**}**, If[UnevaluatedQ[Definition2, x], False, If[SysFuncQ[x], False, a =**

Definition2[x][[If[Options[x] == {}, 1 ;; –2, 1 ;; –3]]]; For[k, k <= Length[a], k++, p = c <> ToString[k]; ToExpression[p <> a[[k]]]; AppendTo[d, If[QFunction[b = p <> ToString[x]], True, False]]; ToExpression["ClearAll[" <> b <> "]"]]; Clear[c]; If[DeleteDuplicates[d] == {True}, True, False]]]] In[2643]:= **F[x_] := x^2; F[x_, y_] = x + y; F := Compile[{{x, _Real}, {y, _Real}}, (x + y)^2]; F = Compile[{{x, _Real}, {y, _Real}}, (x + y)^2];**

In[2644] **:= Map[QFunction1, {"Sin", "F", "Art", "V", "Kr", "GS", "Sv", "S"}]**
Out[2644]= {False, True, False, True, False, True, True, True}
In[2645]:= **G[x_] := x; SetAttributes[G, Protected];**

{ **QFunction[G], QFunction1["G"]**} Out[2645]= {True, True}
In[2646]:= {**Map[QFunction, {Art, Kr}], Map[QFunction1, {"Art", "Kr"}]**}
Out[2646]= {{False, False}, {False, False}}
In[2647]:= **Sv[x_] := x; Sv[x_, y_] := x+y;** {**QFunction[Sv], QFunction1["Sv"]**}
Out[2647]= {False, True}

However, the **QFunction** procedure, successfully testing functional objects which are determined both by the traditional functions with headings and generated by the standard**Compile** function doesn't process*pure* functions; at that, this procedure doesn't process also the*functional* objects of the same name as visually illustrate the last example of the previous fragment. While the**QFunction1** procedure solves the given problem, whose source code is represented in the second part of the previous fragment. The procedure call **QFunction1[*x*]** returns*True* on a traditional function***x*** and an object***x*,** that has been generated by the**Compile** function, and*False* otherwise; moreover, on an object***x*** of the same name*True* is returned only if all its components are traditional functions and/or are generated by the**Compile** function. At that, the call**QFunction1[*x*]** assumes coding of factual***x*** argument in string format. Both procedures enough effectively process options and attributes of the tested objects. Meanwhile, both the**QFunction1** procedure, and the **QFunction** procedure can't correctly test, generally speaking,*pure* functions as quite visually illustrate examples of the previous fragment.

Along with the above types of functions the **Mathematica** system uses also the**Compile** function intended for compilation of functions which calculate *numerical* expressions at certain assumptions. The**Compile** function has the following four formats of coding, each of which is oriented on separate type of compilation, namely:

Compile[{***x1, x2, …***}**,*J*]**–*compiles a function for calculation of an expression**J**in the assumption that all values of arguments**x**$_j$ {**j=1,2,…**}have numerical character;*
Compile[{{***x1, t1***}, {***x2, t2***}, {***x3, t3***}, …}**,*J*]**–*compiles a function for calculation of an expression**J**in the assumption that all values of arguments**xj**have accordingly type**tj** {**j= 1, 2, 3, …**};*
Compile[{{***x1,p1,w1***}, {***x2,p2,w2***}, …}**,*J*]**–*compiles a function for calculation of an expression**J**in the assumption that values of arguments**xj**are ranks**wj**of an array of objects,each of which corresponds to a**pj**type {**j= 1, 2, 3, …**};* **Compile[*s*,*J*,** {{***p1,pw1***}, {{***p2,pw2***}, …}**]**–*compiles a function for calculation of an expression**J**in the assumption that its subexpressions**s**which correspond to the **pj**templates have the**pw**$_j$types accordingly {**j = 1, 2, 3, …**}.*

The **Compile** function processes procedural and functional objects, matrix operations, numerical functions, functions of work with lists, etc.**Compile** function generates a special object**CompiledFunction.** The call**Compile[…, Evaluate[*exp*]]** is used to specify that***exp*** should be evaluated symbolically before compilation.

For testing of this type of functions a rather simple **CompileFuncQ** function can be supposed whose call**CompileFuncQ[*x*]** returns*True* if***x*** represents a **Compile** function, and*False* otherwise. The following fragment represents source code of the function with the most typical examples of its usage.

In[2367] **:= V := Compile[{{x, _Real}, {y, _Real}}, x*y^2]; Kr := (#1*#2^4) &; Art := Function[{x, y}, x*Sin[y]]; H[x_] := Block[{}, x]; H[x_, y_] := x + y; SetAttributes["H", Protected]; P[x__] := Plus[Sequences[{x}]]; GS[x_ /; IntegerQ[x], y_ /; IntegerQ[y]] := Sin[78] + Cos[42]; Sv[x_ /; IntegerQ[x], y_ /; IntegerQ[y]] := x^2 + y^2; Sv = Compile[{{x, _Integer}, {y, _Real}}, (x + y)^6];**

S := Compile[{{x, _Integer}, {y, _Real}}, (x + y)^3]; G = Compile[{{x, _Integer}, {y, _Real}}, (x + y)]; P[x_] := Module[{}, x]

In[2368] **:= CompileFuncQ[x_] := If[SuffPref[ToString[InputForm[Definition2[x]]], "Definition2[CompiledFunction[{", 1], True, False]**

In[2369] **:= Map[CompileFuncQ, {Sv, S, G, V, P, Art, Kr, H, GS, ProcQ}]** Out[2369]= {True, True, True, True, False, False, False, False, False, False} In[2370]**:= Map[CompileFuncQ, {80, avz, a + b, Sin, While, 42.72}]** Out[2370]= {False, False, False, False, False, False}
The**CompileFuncQ** procedure expands possibilities of testing of functional objects in the**Mathematica** system, representing quite certain interest, first of all, for problems of system programming.
The**PureFuncQ** function presented below is oriented for testing of the pure functions, its call**PureFuncQ[*f*]** returns*True* if***f*** defines a*pure* function, and *False* otherwise. The fragment represents source code of the function along with examples of its typical usage.

In[2385] **:= PureFuncQ[f_] := Quiet[StringTake[ToString[f], {–3,–1}] == " & " && ! StringFreeQ[ToString[f], "#"] || SuffPref[ToString[InputForm[f]], "Function[", 1]]**

In[2386] **:= Map[PureFuncQ, {#1 + #2 &, Function[{x, y, z}, x+y], G, ProcQ}]**
Out[2386]= {True, True, False, False}
In[2387]**:= Map[PureFuncQ, {Sin, F, Art, V, Kr, GS, Sv, S}]**
Out[2387]= {False, False, True, False, True, False, False, False} In[2388]**:= Z := Function[{x, y, z}, x + y + z]; SetAttributes[Z, Protected]** In[2389]**:= {PureFuncQ[Z], Attributes[Z]}**
Out[2389]= {True, {Protected}}

In[2390]**:= FunctionQ[x_] := If[StringQ[x], PureFuncQ[ToExpression[x]]|| QFunction1[x], PureFuncQ[x]||QFunction[x]**

In[2391] **:= Map[FunctionQ, {"G", "ProcQ", "Function[{x, y, z}, x + y*z] ", "#1 + #2*#3 &"}]**
Out[2391]= {True, False, True, True}

```
In[2392]:= Map[FunctionQ, {"V","S","Art","Kr","Pi","42.72","GS","Sv","F"}]
Out[2392]= {True, True, True, True, False, False, True, True, True}
In[2393]:= Map[QFunction, {V, S, Art, Kr, Pi, 42.72, GS, Sv, F}]
Out[2393]= {True, True, False, False, False, False, True, True, True}
```

The simple enough **FunctionQ** function completes the previous fragment; its call**FunctionQ[*x*]** returns*True* if an object***x*** is a function of any type of both traditional, and pure, and*False* otherwise. In addition, the name***x*** of an object can be coded both in symbolical, and in string formats; in the second case correct testing of an object***x*** is supported, permitting multiplicity of its definitions, i.e. the object***x*** can be of the same name in the above-mentioned sense. It must be kept in mind that the means of testing that are represented above refers to the testing means of the user functions, and aren't intended for standard functions of the**Mathematica** system, returning on them, as a rule,*False.* So, a number of means for the differentiated identification of the user functions of and traditional, and pure functions has been determined, in particular, procedures and functions**FunctionQ, QFunction, QFunction1** and**PureFuncQ** respectively. Thus, these means provide strict*differentiation* of such basic element of*functional* and*procedural* programming, as***function.*** These and means similar to them are useful enough in applied and system programming in the environment of the**Mathematica** system.

Meantime, here it is necessary to make one very essential remark once again. As it was already noted, unlike the majority of the known languages***Math***language identifies procedures and functions not on their names, but on the headings, allowing not only the procedures of the same name with different headings,but also their combinations with functions. Therefore the question of testing of program objects in context of type {***Procedure, Function***} isn't so unambiguous. The means of testing presented above {**ProcQ, QFunction1, FunctionQ,PureFuncQ,***etc*.} allow as argument***x*** an object or only with one heading or the*first* object returned by the system call**Definition[*x*]** as it was illustrated above. At that, for on objects of the same name the calls of a series of means, considered above return*True* only in a case when the definitions composing them are associated with subobjects of the same type.

In this connection it is very expedient to define some testing procedure that determines belonging of an object***x*** to a group {**Block, CompiledFunction, Function, Module, PureFunction, ShortPureFunction**}. As one of similar approaches it is possible to offer procedure, whose call**ProcFuncTypeQ[*x*]** returns the list of format {*True,* {*t1,t2,…,tp*}} if a simple object***x*** or subobjects of an object***x*** of the same name whose name***x*** is coded in string format have the types***tj*** from the set {*CompiledFunction,PureFunction,ShortPureFunction, Block,Function,Module*}, otherwise the list of format {*False, **x**, "Expression"*} or {*False, **x**, "System"*} is returned. In the case of an object***x*** of the same name a sublist of types {***t1, t2, …, tp***}***(j=1..p)*** of subobjects composing***x*** is returned; whereas***"System"*** and***"Expression"*** determines a system function***x*** and an expression***x*** respectively. So, the**ProcFuncTypeQ** procedure can be applied as a group test for belonging of an object***x*** to the above types. The following fragment represents source code of the**ProcFuncTypeQ** procedure with the most typical examples of its usage.

```
In[2528]:= ProcFuncTypeQ[x_ /; StringQ[x]] := Module[{a, b, d = {}, k = 1, p},
```

**If[ShortPureFuncQ[x], {True, “ShortPureFunction”},
If[SuffPref[x, “Function[{“, 1], {True, “PureFunction”},
If[UnevaluatedQ[Definition2, x], {False, x, “Expression”},**

**If[SysFuncQ[x], {False, x, “System”},
a = Definition2[x][[If[Options[x] == {}, 1 ;;–2, 1 ;;–3]]]; For[k, k <= Length[a], k++,
b = Flatten[{HeadPF[x]}]; b = Flatten[Map[Map3[StringJoin, #, {” := “, ” = “}] &,
b]]; p = StringReplace[a[[k]], GenRules[b, ””], 1];**

**If[SuffPref[p, {“Compile[{“, “CompiledFunction[{”}, 1],
AppendTo[d, “CompiledFunction”], If[SuffPref[p, “Block[{“, 1], AppendTo[d,
“Block”],
If[SuffPref[p, “Module[“, 1], AppendTo[d, “Module”], If[SuffPref[p, “Function[{“,
1], AppendTo[d, “PureFunction”], If[ShortPureFuncQ[p], AppendTo[d,
“ShortPureFunction”], If[PureFuncQ[ToExpression[x]], AppendTo[d,
“PureFunction”], AppendTo[d, “Function”]]]]]]]]; {True, d}]]]]]**

In[2529] := **V := Compile[{{x, _Real}, {y, _Real}}, (x^3 + y)^2]; Sv[x_] :=
Module[{}, x]; Art := Function[{x, y}, x*Sin[y]]; GS[x_ /; IntegerQ[x], y_ /;
IntegerQ[y]] := Sin[x] + y; Sv[x_ /; IntegerQ[x], y_ /; IntegerQ[y]] := x^2 + y^2; Sv =
Compile[{{x, _Integer}, {y, _Real}}, (x + 6*y)^6];**

**S := Compile[{{x, _Integer}, {y, _Real}}, (x + y)^3]; Kr := (#1*#2^4) &; G =
Compile[{{x, _Integer}, {y, _Real}}, (x + y)]; H[x_] := Block[{a}, x]; H[x_, y_] := x +
y; SetAttributes[“H”, Protected]**

In[2530] **:= ProcFuncTypeQ[“Sv”]**
Out[2530]= {True, {”CompiledFunction“, “Module“, “Function”}} In[2531]**:=
ProcFuncTypeQ[“H”]**
Out[2531]= {True, {”Block“, “Function”}}
In[2532]**:= ProcFuncTypeQ[“G”]**
Out[2532]= {True, {”CompiledFunction”}}

In[2533] **:= A[x_, y_] := x + y; A[x_] := x; ProcFuncTypeQ[“A”]** Out[2533]= {True,
{”Function“, “Function”}}
In[2534]**:= ProcFuncTypeQ[“Art”]**
Out[2534]= {True, {”PureFunction”}}
In[2535]**:= ProcFuncTypeQ[“Function[{x, y}, x+y]”]**
Out[2535]= {True, “PureFunction”}
In[2536]**:= ProcFuncTypeQ[“Kr”]**
Out[2536]= {True, {”ShortPureFunction”}}
In[2537]**:= ProcFuncTypeQ[“a+g+s*#&”]**
Out[2537]= {True, “ShortPureFunction”}
In[2538]**:= ProcFuncTypeQ[“GS”]**
Out[2538]= {True, {”Function”}}
In[2539]**:= ProcFuncTypeQ[“S”]**
Out[2539]= {True, {”CompiledFunction”}}

In[2543]**:= ShortPureFuncQ[x_] :=
PureFuncQ[ToExpression[If[StringQ[x], x, ToString[x]]]] &&**

StringTake[StringTrim[ToString[If[StringQ[x], ToExpression[x], ToString[x]]]], {–1,–1}] == "&"

In[2544] **:= Map[ShortPureFuncQ, {"a+g+s*#&", Kr, "Kr", a + g + s*# &}]**
Out[2544]= {True, True, True, True}
In[2545]**:= ProcFuncTypeQ["2015"]**
Out[2545]= {False, "2015", "Expression"}
In[2546]**:= ProcFuncTypeQ["While"]**
Out[2546]= {False, "While", "System"}

Along with the **ProcFuncTypeQ** procedure the above fragment represents a simple and useful enough function, whose call**ShortPureFuncQ[*x*]** returns *True* if***x*** determines a pure function in short format, and*False* otherwise. In, particular, this function is used by the**ProcFuncTypeQ** procedure too. In a number of applications of both the applied, and system character which are connected with*processing* of procedures and functions the**ProcFuncTypeQ** procedure is a rather effective testing group means.

For identification of functional objects *(**traditional**and**pure**functions)* in the **Mathematica** system exist quite limited means that are based only on calls of the system functions**Part[*x*, *0*]** and**Head[*x*]** which return the headings of an expression***x;*** at that, on pure functions**Function** is returned, whereas on the traditional functions**Symbol** is returned as a simple enough fragment a rather visually illustrates, namely**:**

In[2905] **:= G[x_Integer, y_Real, z_Real] := x*y^2 + z**
In[2906]**:= Map[Head, {#1*#2*#3 &, Function[{x, y, z}, x + y*z], G, ProcQ}]**
Out[2906]= {Function, Function, Symbol, Symbol}
In[2907]**:= Mapp[Part, {#1*#2*#3 &, Function[{x, y, z}, x*y*z], G, ProcQ}, 0]**
Out[2907]= {Function, Function, Symbol, Symbol}
In[2908]**:= Map[PureFuncQ, {#1*#2 &, Function[{x, y, z}, x*y*z], G, ProcQ}]**
Out[2908]= {True, True, False, False}
In[2909]**:= Map[QFunction, {#1*#2 &, Function[{x, y, z}, x/y*z], G, ProcQ}]**
Out[2909]= {False, False, True, False}
In[2910]**:= Map[FunctionQ, {#1*#2 &, Function[{x, y, z}, x*y*z], G, ProcQ}]**
Out[2910]= {True, True, True, False}
In[2911]**:= {m, n}= {#1 + #2*#3 &, Function[{x, y}, x*y]}; Map[Head, {m, n}]**
Out[2911]= {Function, Function}
In[2912]**:= {Mapp[Part, {m, n}, 0], Map[QFunction, {m, n}]}**
Out[2912]= {{Function, Function}, {False, False}}
In[2913]**:= {Map[FunctionQ, {m, n}], Map[PureFuncQ, {m, n}]}** Out[2913]= {{True, True}, {True, True}}

In this context the **Head2** procedure seems as an useful enough means that is a modification of the**Head1** procedure and that is based on the previous **ProcFuncTypeQ** procedure and the standard**Head** function. The procedure call**Head2[*x*]** returns the heading or the type of an object***x*,** given in string format. In principle, the*type* of an object can quite be considered as a*heading* in its broad understanding. The**Head2** procedure serves to such problem generalizing the standard**Head** function and returning the heading of an expression*x* in the context of {***Block, CompiledFunction, Function, Module, PureFunction, ShortPureFunction, Symbol, System,*****Head[*x*]**}. The examples of use of both means on the same list of the tested objects that in a number of cases confirm preference of the**Head2** procedure are given as a comparison. The following fragment represents source code of the**Head2** procedure and the most typical examples of its use. Whereas the**Head3** function presented here expands the system function**Head** and our procedures**Head1, Head2** upon condition, that a tested expression***x*** is considered apart from the sign**;** distinction is visually illustrated by results of the call of these means on the identical actual arguments. In general, the function call**Head3[*x*]** is similar to the procedure call**Head1[*x*].**

In[2651]**:= Head2[x_] := Module[{b,
a = Quiet[Check[ProcFuncTypeQ[ToString[x]], {Head[x]}]]}, If[SameQ[a[[–1]], "System"], "System",
If[SameQ[a[[–1]], "Expression"], Head[x], If[ListQ[a], b = a[[–1]]]; If[Length[b] == 1, b[[1]], b]]]]**

In[2652] **:= Map[Head2, {"#1 + #2*#3&", "Function[{x, y, z}, x+y*z]", "G", "ProcQ", "a + b", "{x, y, z}", ""ransian"", Avz, While}]**
Out[2652]= {"ShortPureFunction"**,** "PureFunction"**,** {"Block"**,** "Block"**,** "Module"}**,** "Module"**,** String**,** String**,** String**,** Symbol**,** "System"}
In[2653]**:= Map[Head2, {"V", "Art", "G", "ProcQ", "GS", "Sv", "S", "H", "Agn", 80, 42.47, Kr}]**
Out[2653]= {"CompiledFunction"**,** "PureFunction"**,** {"CompiledFunction"**,** "Function"}**,** "Module"**,** "Function"**,** {"CompiledFunction"**,** "Module"**,** "Function"}**,** "CompiledFunction"**,** {"Block"**,** "Function"}**,** String**,** Integer**,** Real**,** "ShortPureFunction"}
In[2654]**:= Map[Head, {"V", "Art", "G", "ProcQ", "GS", "Sv", "S", "H", "Agn", 80, 42.47, Kr}]**
Out[2654]= {CompiledFunction**,** Function**,** CompiledFunction**,** Symbol**,** Symbol**,** CompiledFunction**,** CompiledFunction**,** Symbol**,** Symbol**,** Integer**,** Real**,** Function}

In[2655]**:= Head3[x_] := Symbol[If[Part[x, 1] ===–1, Head1[–1*x], Head1[x]]]**
In[2656]**:=** {**Head[Sin[–a + b]], Head2[Sin[–a + b]], Head3[Sin[–a + b]]**} Out[2656]= {Times**,** Times**,** Sin}

At last, quite natural interest represents the question of existence of the user procedures and functions activated in the current session. The solution of the given question can be received by means of the procedure whose procedure call**ActBFMuserQ[]** returns*True* if such objects in the current session exist, and*False* otherwise**;** meanwhile**,** meanwhile, the call**ActBFMuserQ[*x*]** thru optional argument***x* –** an undefinite variable**–** returns the**2–**element nested list whose the first element contains name of the user object in string format while the second defines list of its types in string format respectively. The fragment presents source code of the**ActBFMuserQ** and examples of its use.

In[2570] **:= ActBFMuserQ[x___ /; If[{x}== {}, True, If[Length[{x}] == 1 && ! HowAct[x], True, False]]] := Module[{b = {}, c = 1, d, h, a = Select[Names["`*"], ! UnevaluatedQ[Definition2, #] &]}, For[c, c <=Length[a], c++, h = Quiet[ProcFuncTypeQ[a[[c]]]];**

If[h[[1]], AppendTo[b, {a[[c]], h[[–1]]}], Null]]; If[b == {}, False, If[{x}!= {}, x =

```
If[Length[b] == 1, b[[1]], b]]; True]]
```

```
In[2571] := V := Compile[{{x, _Real}, {y, _Real}}, (x^3 + y)^2];
Art := Function[{x, y}, x*Sin[y]]; Kr := (#1^2 + #2^4) &; GS[x_ /; IntegerQ[x], y_ /;
IntegerQ[y]] := Sin[80] + Cos[42];
```

```
G = Compile[ {{x, _Integer}, {y, _Real}}, x*y]; P[x_, y_] := Module[{}, x*y] H[x_] :=
Block[{}, x]; H[x_, y_] := x + y; SetAttributes["H", Protected] P[x_] := Module[{},
x]; P[y_] := Module[{}, y];
P[x__] := Plus[Sequences[{x}]]; P[x___] := Plus[Sequences[{x}]]; P[y_ /; PrimeQ[y]]
:= Module[{a = "Agn"}, y]; T42[x_, y_, z_] := x*y*z P[x_ /; StringQ[x]] :=
Module[{}, x]; P[x_ /; ListQ[x]] := Module[{}, x]; R[x_] := Module[{a = 500}, x*a];
GSV := (#1^2 + #2^4 + #3^6) &
```

```
In[2572] := ActBFMuserQ[]
Out[2572]= True
In[2573]:= {ActBFMuserQ[t], t}
Out[2573]= {True, {{"Art", {"PureFunction"}}, {"G", {"CompiledFunction"}},
{ "GS", {"Function"}}, {"H", {"Block", "Function"}}, {"P1", {"Function"}}, {"W",
{"Function"}}, {"R", {"Module"}}, {"Kr", "ShortPureFunction"}, {"V",
{"CompiledFunction"}}, {"P", {"Module", "Module", "Module", "Module", "Module",
"Function", "Function"}, {"T42", {"Function"}}, {"GSV", "ShortPureFunction"}}}}
```

At that, the given procedure along with standard means uses and our means such as**Definition2, HowAct, ProcFuncTypeQ** and**UnevaluatedQ** that are considered in the present book and in [28-33]. The procedure is represented as a rather useful means, first of all, in the system programming.

6.4. Headings of procedures and functions in the *Mathematica*system

In many contexts, it is not necessary to know the exact value of an arbitrary expression**;** it suffices to know that the given expression belongs to a certain broad class, or group, of expressions which share some common properties. These classes or groups are known as***types.*** If***T*** represents a type, then an expression is of type***T*** if it belongs to the class that***T*** presents. For example, an expression is said to be of type***Integer*** if it belongs to the definite class of expressions denoted by the type name***Integer*,** which is the set of integers. Many procedures and functions use the types to direct the flow of control in algorithms or to decide whether an expression is a valid input. For example, the behavior of a function or a procedure generally depends on the types of its actual arguments. At that, the result of a number of operations is defined by type of their arguments. The***type* –** fundamental concept of the theory of programming, defining an admissible set of values or operations which can be applied to such values and, perhaps, also a way of realization of storage of values and performance of operations.

Any objects with which the programs operate, belong to certain types. The concept of data type in programming languages of high level appeared as absolutely natural reflection of that fact that the data and expressions which are processed by a program can have various sets of admissible values, be stored in RAM of the computer in different ways, be processed by different commands of the processor. At that, the type of any object can be defined in ***2*** ways, namely**:** by a set of all values belonging to this type, or by a certain predicate function defining object belonging to this type. Advantages from use of types of objects are reduced to three highlights**:*(1)*** protection against errors of assignment, incorrect operations along with inadmissible factual arguments passed to a procedure/function**;*(2)*** the standardization provided by agreements on the types supported by the majority of the programming systems,***(3)*** documenting of software in many respects becomes simpler at use of the standard typification of the objects and data used in them. In the modern languages of programming are used several systems of typification a brief characteristic of which can be found in [33] whereas more in detail it is possible to familiarize oneself with them in a number of books on modern programming systems. Considering importance of typification of language objects, this aspect is considered by us and concerning**Mathematica** system in books [28-33]. In particular, in our book [30] enough in details from point of view of development of the mechanism of typification both**Mathematica** and**Maple** systems are considered as computer mathematics systems which are the most developed and popular for today.

Unlike ***209*** types, for example, of the***Maple 11*** which are tested by the***`type`*** procedure*(apart from a considerable enough set of the user types connected to the **Maple**by means of our library* [47])**,** the***Mathematica 9*** has only***60*** testing***Q–*** functions whose names have the form***NameQ,*** for example, call**SyntaxQ[*x*]** returns*True* if the contents of string***x*** is a correct***Mathematica–***expression, and*False* otherwise. In a certain measure to this function the**ToExpression** function adjoins that evaluates all expressions which are in the argument of string format with return of*Null****.*** By results of their performance the given functions can be considered as testing tools of correctness of the expressions that are in a string. At that, if in the first case we receive value {*True***,***False*}**,** in the second case the correctness can be associated with return of*Null*. In this context the**ToExpression** function in a certain relation is similar to the ***`parse`*** procedure of the**Maple** system [10,14-16,21,25-27]. If necessary, the user can also create own functions with names of the form***NameQ*** which will significantly expand the range of similar standard system tools. Below, this question largely is detailed on specific examples of such means.

Coding of definitions of types directly in headings of procedures/functions takes place only for the**Mathematica** system, allowing in the call point of a procedure/function without execution of it and without appeal to external tools to execute testing for an admissibility of the actual arguments received by the procedure/function. Such approach increases efficiency of execution of a procedure/function, doing it by the more mobile. The given approach is especially convenient in the case where the type posesses highly specialized character or its definition is described by small and a rather clear program code. Indeed, in a number of cases the inclusion of definitions of the testing means directly into headings is very conveniently. So, this approach is used rather widely in means from the***AVZ_Package*** package [48]. In general, this approach allows to typify quite in details in many important applied data**;** its essence rather visually illustrates the following simple fragment, namely**:**

```
In[2524] := ArtKr[x_ /; {T[z_] := If[z <= 80 && z >= 8, True, False], T[x]}[[2]], y_ /;
StringQ[y] && ! SuffPref[y, {"avz", "agn", "vsv"}, 1]] := Module[{a = 72, b = 67}, y
```

<> " = " <> ToString[x + a + b]]

In[2525] **:=** {**T[6], Map7[ArtKr, Sequences,** {{**72, "h"**}, {**42, "j"**}, {**50, "avagvs"**}}**], T[6]**}
Out[2525]= {T[6]**,** {"h= 211"**,** "j= 181"**,** "avagvs= 189"}**,** False}
In[2526]**:= Definition[T]**
Out[2526]= T[z_]**:=** If[z<= 80**&&** z>= 8**,** True**,** False]

The example of simple ***ArtKr*** procedure of the modular type quite visually illustrates opportunities on the organization of typified testing of the actual arguments of the procedure when definition of a type***T*** is given directly in heading of the procedure and is activated at once after the first call of***ArtKr*** procedure. Many of means of***AVZ_Package*** package use similar approach in own organization [48].

On the assumption of the general definition of a procedure, in particular, of modular type
***M*[*x_/*;*Test*$_x$,*y_/*;*Test*$_y$, …] := Module[**{*locals*}**,*Procedure body*]**

and of that fact that concrete definition of the procedure is identified not by its***name*,** but its***heading*** we will consider a set of the useful enough means which provide the various manipulations with headings of procedures and functions, and play a very important part in procedural programming and, first of all, programming of problems of the system character. Having defined such object rather useful in many appendices as the*heading* of a procedure/function in the form**"***Name*[*The list of formal arguments with the testing means ascribed to them*]**",** quite naturally arises the question of the creation of means for a testing of objects regarding their relation to the type **`*Heading*`.** It is possible to represent the**HeadingQ** procedure as a such tool whose source code with examples of its use is represented by the following fragment. The procedure call**HeadingQ[*x*]** returns*True* if an object***x*,** given in string format, can be considered as a syntactic correct heading**;** otherwise *False* is returned**;** in case of inadmissible argument***x*** the call**HeadingQ[*x*]** is returned unevaluated. The**HeadingQ** procedure is rather essentially used in a series of means from the***AVZ_Package*** package [48].

In[3385]**:= HeadingQ[x_ /; StringQ[x]] := Module[**{**a, b, c, k = 1, m = True, n = True**}**, If[StringTake[x,** {**–1,–1**}**] == "]" &&**

StringCount[x, {**"[", "]"**}**] == 2 && ! StringFreeQ[StringReplace[x, " "–> ""], "[]"], Return[m], If[! StringFreeQ[RedSymbStr[x, "_", "_"], "[_]"], Return[! m]]]; Quiet[Check[ToExpression[x], Return[False]]]; If[DeleteDuplicates[Map3[StringFreeQ, x,** {**"[", "]"**}**]] ===** {**False**}**, c = StringPosition[x, "["][[1]][[2]]; If[c == 1, Return[False], a = StringTake[x,** {**c,–1**}**]], Return[False]]; b = StringPosition[a, "["][[1]][[1]]; c = StringPosition[a, "]"][[–1]] [[1]]; a = "**{**" <> StringTake[a,** {**b + 1, c–1**}**] <> "**}**"; a = Map[ToString, ToExpression[a]];**

If[DeleteDuplicates[Mapp[StringFreeQ, a, "_"]] =={**False**}**, Return[True]]; If[**{**c, a**}**==** {**2,** {}}**, Return[True], If[a ==** {}**|| StringTake[a[[1]],** {**1, 1**}**] == "_", Return[False], For[k, k <= Length[a], k++, b = a[[k]];**

If[StringReplace[b, "_"–> ""] != "" && StringTake[b, {**–1,–1**}**] == "_" || ! StringFreeQ[b, "_ "] || ! StringFreeQ[b, "_:"]||! StringFreeQ[b, "_."], m = True, n =**

False]]]; m && n]]

In[3386] := {**HeadingQ["D[x_, y_/; ListQ[y], z_:75, h_]"], HeadingQ["D[x_, y_, z_:75, h_]"], HeadingQ["D[x_, y_/; ListQ[y], z_:75, _]"]**}

Out[3386]= {True, True, True}
In[3387]= {**HeadingQ["D[x_, y_/; ListQ[y], z_:75, h]"],**

HeadingQ["[x_, y_/; ListQ[y], z:75]"] }
Out[3387]= {False, False}
In[3388]:= {**HeadingQ["g[]"], HeadingQ["t[x__]"], HeadingQ["p[x__]"],**

HeadingQ["h[_]"] }
Out[3388]= {True, True, True, False}
In[3389]:= {**HeadingQ["D[_, x_]"], HeadingQ["Z[x__]"],**

HeadingQ["Q[x___]"]}
Out[3389]= {True, True, True}

In[3390] := {**HeadingQ["D[x_, y_/; ListQ[y], z_:75, h]"], HeadingQ["V[x_, y_/;ListQ[y], z_.]"]**}
Out[3390]= {False, True}

At that, the given procedure along with standard means uses and our means such as**RedSymbStr, Map3** and**Mapp** which are considered in the present book and in [28-33]. The procedure is represented as a rather useful means, first of all, in system programming, for example, at testing of objects types in definitions of procedures and functions similarly to the means**Head1** and **Head2,** considered in the present book too.
The following**HeadingQ1** procedure represents a very useful expansion of the above**HeadingQ** procedure concerning its opportunity of testing of the headings onto their correctness. The procedure call**HeadingQ1[*x*]** returns *True* if the actual argument***x*,** given in string format, can be considered as a syntactically correct heading**;** otherwise*False* is returned. The next fragment represents source code of the**HeadingQ1** procedure and examples of its use.

In[2512] := **HeadingQ1[x_ /; StringQ[x]] := Module[**{**b, c =** {}**, d, h = "F", k = 1, a = Quiet[StringTake[x,** {**Flatten[StringPosition[x, "[", 1]][[1]]+ 1,–2**}**]]**}**, If[StringFreeQ[x, "["], False, b = StringSplit1[a, ","]; For[k, k <= Length[b], k++, d = b[[k]]; c = Append[c, If[StringFreeQ[d, "_"], False, If[MemberQ[ToString /@** {**Complex, Integer, List, Rational, Real, String, Symbol**}**, StringTake[d,** {**Flatten[StringPosition[d, "_"]][[–1]] + 1,–1**}**]], True, HeadingQ[h <> "[" <> d <> "]"]]]]]; If[DeleteDuplicates[c] ==** {**True**}**, True, False]]]**

In[2513] := **Map[HeadingQ1,** {**"H[s_String,x_/;StringQ[x],y_]", "T[x_,y_/;ListQ[y],z_List]", "V[x_, y_/; ListQ[y]&&Length[L] == 90]", "E[x__, y_/; ListQ[y], z___]"**}**]**

Out[2513] = {True, True, True, True}
In[2514]:= {**Map[HeadingQ,** {**"H[s_Integer]", "G[n_Integer,L_List]", "G[n___Integer]"**}**], Map[HeadingQ1,** {**"H[s_Integer]", "G[n_Integer,L_List]", "G[n___Integer]"**}**]**}

Out[2514]= {{True, True, True}, {True, True, True}}
In addition to the system means the**HeadingQ1** procedure uses procedure **StringSplit1** that represents an useful generalization of the system function **StringSplit.** It should be noted that regardless of the correct testing of quite wide type of headings, meanwhile, the procedure**HeadingQ** along with the **HeadingQ1** not has comprehensive character because of a series of features of syntactical control of***Math*–**language. That the following simple example very visually illustrates, from which follows, that the system testing means perceive incorrect headings as correct expressions.

In[2567] **:= ToExpression["W[x__/;_StringQ[x]]"]**
Out[2567]= W[x__/; _StringQ[x]]
In[2568]**:= SyntaxQ["W[x__/;_StringQ[x]]"]**
Out[2568]= True

At the same time two these procedures are rather useful in many cases. Meanwhile, on the basis of our**ArgsTypes** procedure serving for testing of formal arguments of a function/procedure which has been activated in the current session perhaps further expansion of the testing opportunities of the **HeadingQ1,** allowing in certain cases to expand types of the correctly tested headings of procedures/functions. Meanwhile, here it is possible to tell only about expansion of opportunities at certain cases, but not about expansion as a whole. The fragment below represents source code of the**HeadingQ2** procedure along with the most typical examples of its usage.
In[2942]**:= HeadingQ2[x_ /; StringQ[x]] :=**

Module[{a, b, c, d = ToString[Unique["agn"]]}, {a, b}= Map[DeleteDuplicates, Map[Flatten, Map3[StringPosition, x, {"[", "]"}]]]; If[StringLength[x] == b[[–1]] && SymbolQ[c = StringTake[x, {1, a[[1]]–1}]], Quiet[Check[ToExpression[

StringReplace[x, c <> "[" –> d <> "[", 1] <> " := 72"], False]]; c = Map[SyntaxQ, ArgsTypes[d]]; ToExpression["Remove[" <> d <> "]"]; If[DeleteDuplicates[c] === {True}, True, False], False]]

In[2943]**:= Map8[HeadingQ1, HeadingQ2, {"V[x__/_String]"}]**

Out[2943] = {True, False}
In[2944]**:= Map8[HeadingQ1, HeadingQ2, {"V[x_/; StringQ[x]]"}]** Out[2944]= {True, True}
In[2945]**:= Map[HeadingQ2, {"F[x_/; StringQ[x]]", "F[x/; StringQ[x]]",**

"F[x; StringQ[x]]", "F[x_ /_ StringQ[x]]", "F[x_//; StringQ[x]]", "F[x_; y_; z_]"}]
Out[2945]= {True, True, True, False, False, True}
In[2946]**:= Map[HeadingQ1, {"F[x_/; StringQ[x]]", "F[x/; StringQ[x]]",**
"F[x; StringQ[x]]", "F[x_/_ StringQ[x]]",
"F[x_//; StringQ[x]]", "F[x_; y_; z_]"}] Out[2946]= {True, False, False, True, False, True}
In[2947]**:= Map[HeadingQ, {"F[x_/; StringQ[x]]", "F[x/; StringQ[x]]", "F[x; StringQ[x]]", "F[x_/_ StringQ[x]]", "F[x_//; StringQ[x]]", "F[x_; y_; z_]"}]**
Out[2947]= {True, False, False, True, False, True}
In[2948]**:= Map[#["F[x_/_ StringQ[x]]"] &, {HeadingQ, HeadingQ1}]** Out[2948]= {True, True}

In[2949]**:= Map[#["F[x_/_ StringQ[x]]"] &,** {**HeadingQ2, HeadingQ3**}**]** Out[2949]= {False**,** False}
In[2950]**:= Map[#["F[x_/_StringQ[x]]"] &,** {**HeadingQ, HeadingQ1**}**]** Out[2950]= {True**,** True}
In[2951]**:= Map[#["F[x_/_StringQ[x]]"] &,** {**HeadingQ2, HeadingQ3**}**]** Out[2951]= {False**,** False}

Analogously to the procedures **HeadingQ** and**HeadingQ1,** the procedure call**HeadingQ2[*x*]** returns*True* if actual argument***x*,** set in string format, can be considered as a syntactically correct heading**;** otherwise*False* is returned. At that, the examples presented in the above fragment of applications of the procedures**HeadingQ, HeadingQ1** and**HeadingQ2** quite visually illustrate distinctions between their functionality. The group of these means includes also the**HeadingQ3** procedure that in the functional relation is equivalent to the**HeadingQ2** procedure**;** its call**HeadingQ3[*x*]** returns*True* if an actual argument***x*,** set in string format, can be considered as a syntactically correct heading**;** otherwise, the call returns*False***.** At the same time between pairs of procedures {**HeadingQ[*x*],HeadingQ1[*x*]**}**&** {**HeadingQ2[*x*],HeadingQ3[*x*]**} principal distinctions exist, in particular, on the headings {**F[*x*_/_StringQ[*x*]], F[*x*_ / _StringQ[*x*]**} the first pair returns*True* while the second pair returns *False* as a quite visually illustrates the above fragment. It is also necessary to note that the first pair of testing functions is more high**–**speed what is very essential at their use in real programming. Meanwhile, considering similar and some other unlikely encoding formats of the headings of functions and procedures**,** the represented four procedures**HeadingQ[*x*], HeadingQ1[*x*], HeadingQ2[*x*]** and**HeadingQ3[*x*]** can be considered as rather useful testing means in modular programming. At that, from experience of their use and their temporary characteristics it became clear that it is quite enough to be limited oneself only by procedures**HeadingQ[*x*], HeadingQ1[*x*]** that cover rather wide range of erroneous coding of the headings. Furthermore, taking into account the mechanism of*parse* of expressions for their correctness that the**Mathematica** system uses, creation of comprehensive tools of testing of the headings is very unlikely. Naturally, it is possible to use non**–**standard receptions for receiving the testing means for the headings having a rather wide set of deviations from the standard, however such outlay do not pay off by the received benefits.

The following procedure serves as an useful enough means at manipulating with procedures and functions, its call**HeadPF[*x*]** returns heading in string format of a block, module or function with a name***x*** activated in the current session, i.e. of function in its traditional understanding with heading. While on other values of argument***x*** the call is returned unevaluated. Meanwhile, the problem of definition of headings is actual also in the case of the objects of the above type of the same name which have more than one heading. In this case the procedure call**HeadPF[*w*]** returns the list of headings in string format of the subobjects composing an object***w*** as a whole. The following fragment represents source code of the**HeadPF** procedure along with the most typical examples of its usage.

In[2942] **:= HeadPF[x_ /; BlockFuncModQ[x]] := Module[**{**b, c= ToString[x], a = Select[Flatten[**{**PureDefinition[x]**}**], ! SuffPref[#, "Default[", 1] &]**}**, b = Map[StringTake[#,** {**1, Flatten[StringPosition[#,** {**" := ", " = "**}**]][[1]]–1**}**] &, a]; If[Length[b] == 1, b[[1]], b]]**

In[2943]:= **G[x_, y_] := x*Sin[y] + y*Cos[x]; s[] := 90*x; g := 500** In[2944]:= **Map[HeadPF, {G, s, Sin, 2015, g}]**

Out[2944] = {"G[x_, y_]", "s[]",HeadPF[Sin], HeadPF[2015], HeadPF[500]} In[2945]:= **Map[HeadPF, {If, Tan, Log, True, G, "Infinity", For, Do, ProcQ}]** Out[2945]= {HeadPF[If], HeadPF[Tan], HeadPF[Log], HeadPF[True],

" G[x_, y_]", HeadPF["Infinity"], HeadPF[For], HeadPF[Do], "ProcQ[x_]"} In[2946]:= **M[x_ /; x == "avzagn"] := Module[{a}, a*x]; M[x_, y_, z_] := x*y*z; M[x_ /; IntegerQ[x], y_String] := Module[{a, b, c}, x]; M[x_String] := x; M[x_, y_] := Module[{a, b, c}, "abc"; x + y];** In[2947]:= **HeadPF[M]**

Out[2947]= {"M[x_ /; x== "avzagn"]", "M[x_, y_, z_]",
"M[x_ /; IntegerQ[x], y_String]", "M[x_String]", "M[x_, y_]"}

So, the call **HeadPF[*x*]** returns the heading in string format of an object with a name***x*** of the type {*block*,*function*,*module*} which has been activated in the current session. At that, for an object***x*** which has several various headings, the call**HeadPF[*x*]** returns the list of the headings whose order fully meets the order of the definitions returned by the function call**Definition[*x*].** In this regard testing of an object***x*** regarding to be of the same name is enough actually; the**QmultiplePF** procedure solves the problem whose source code along with typical examples of its usage the following fragment represents.

In[2780]:= **QmultiplePF[x_, y___] :=
Module[{a = Flatten[{PureDefinition[x]}]}, If[MemberQ[{{"System"}, {$Failed}}, a], False, If[{y}!= {}&& ! HowAct[y], y = If[Length[a] == 1, a[[1]], a]]; True]]**

In[2781] := **M[x_ /; x == "avzagn"] := Module[{a, b, c}, x]; M[x_String] := x; M[x_, y_, z_] := x*y*z; M[x_List, y_] := Block[{a}, Length[x] + y] M[x_ /; IntegerQ[x], y_String] := Module[{a, b, c}, x]; M[x_, y_] := Module[{a, b, c}, "abc"; x + y];**
In[2782]:= **QmultiplePF[M]**

Out[2782] = True
In[2783]:= {**QmultiplePF[M, s], s**}
Out[2783]= {True, {"M[x_ /; x== "avzagn"]:= Module[{a, b, c}, x]",

" M[x_String]:= x", "M[x_, y_, z_]:= x*y*z", "M[x_List, y_]:= Block[{a, b, c}, "abc"; Length[x]+ y]", "M[x_ /; IntegerQ[x], y_String]:= Module[{a, b, c}, x]", "M[x_, y_]:= Module[{a, b, c}, "abc"; x+ y]"}}

In[2784] := **Map[QmultiplePF, {M, 90, Avz, Sin, If, a + b}]**
Out[2784]= {True, False, False, False, False, False}
The procedure call**QmultiplePF[*x*]** returns*True*, if***x*** – an object of the same name*(block*,*function*,*module)*, and*False* otherwise. While the procedure call **QmultiplePF[*x*,*y*]** with the*2nd* optional argument***y*** *–an indefinite variable–* returns through***y*** the list of definitions of all subobjects with a name***x*.** The **QmultiplePF** procedure realization significantly uses the earlier considered **PureDefinition** procedure, and also our**HowAct** function. In a number of cases the**QmultiplePF** procedure despite the relative simplicity is a rather convenient means at testing of objects of the specified types.

At testing of objects often arises necessity of allotment among them of the system functions; this problem is solved by simple function, whose the call **SystemQ[*x*]** returns*True* if an object***x*** is a system function, i.e. is defined by builtin language of the**Mathematica,**and*False* otherwise. The function very simply is defined directly on the basis of the standard functions**Definition, Names** and**ToString.** The following fragment represents source code of the **SystemQ** function with typical examples of its application. In a number of appendices the given function represents quite certain interest and, first of all, giving opportunity quite effectively to differentiate means.

In[2975]**:= SystemQ[S_] := If[Off[Definition::ssle];
! ToString[Definition[S]] === Null && MemberQ[Names["System`*"], ToString[S]], On[Definition::ssle]; True, On[Definition::ssle]; False]**

In[2976] **:= Map[SystemQ, {90, G, Sin, Do, While, False, ProcQ, a/b^2, M}]**
Out[2976]= {False, False, True, True, True, True, False, False, False} Above all, the**SystemQ** function is often used in the headings of procedures and functions, testing the actual arguments for admissibility. In addition to the**SystemQ** function it makes sense to present an useful enough function whose call**LangHoldFuncQ[*x*]** returns*True* if***x* –** a basic function of***Math*–** language, and*False* otherwise. At that, under***basic*** function is understood a system function with one of the attributes ascribed to it, namely**:*HoldFirst, HoldAll*** or***HoldRest.*** The function is represented as a quite useful means in the case of necessity of more accurate differentiation of software. The next fragment represents source code of the**LangHoldFunc** function along with examples of its most typical usage.
In[2299]**:= LangHoldFuncQ[x_] := If[SystemQ[x] &&**

Intersection[Quiet[Check[Attributes[x], False]], {HoldAll, HoldFirst, HoldRest}] != {}, True, False]

In[2300] **:= Map[LangHoldFuncQ, {If, Goto, Do, Sin, Rule, Break, While, Switch, Which, For}]**
Out[2300]= {True, False, True, False, False, False, True, True, True, True}

For a series of problems of system character the **LangHoldFuncQ** function allows to differentiate the set of all system functions of the***Math*–**language according to the specified feature.

Right there pertinently to note some more means linked with the **HeadPF** procedure. So, the**Headings** procedure**–** an useful enough expansion of the **HeadPF** procedure in the case of the blocks/functions/modules of the same name but with various headings. Generally the call**Headings[*x*]** returns the nested list whose elements are the sublists defining respectively headings of subobjects composing an object***x;*** the*first* elements of such sublists defines the types of subobjects whereas others define the*headings* corresponding to them. The next fragment represents source code of the**Headings** procedure along with the most typical examples of its usage.

In[2385] **:= Headings[x_ /; BlockFuncModQ[x]] := Module[{n, d, h, p, t, k =1, c = {{"Block"}, {"Function"}, {"Module"}}, a = Flatten[{PureDefinition[x]}]}, While[k <= Length[a], d = a[[k]]; n = ToString[Unique["agn"]]; ToExpression[n <> d]; ClearAll[p]; h = HeadPF[t = n <> ToString[x]]; d = StringTake[h, {StringLength[n] + 1, –1}]; BlockFuncModQ[t, p]; If[p == "Block", AppendTo[c[[1]], d], If[p == "Function", AppendTo[c[[2]], d], AppendTo[c[[3]], d]]]; ToExpression["Remove[" <>**

t <> "," <> n <> "]"]; k++]; c = Select[c, Length[#] > 1 &]; If[Length[c] == 1, c[[1]], c]] In[2386]:= **M[x_ /; SameQ[x, "avz"], y_] := Module[{a, b, c}, y]; M[x_, y_, z_] := x + y + z; L1[x_, y_] := Block[{a, b, c}, x + y];**

M[x_ /; x == "avz"] := Module[{a, b, c}, x]; L[x_] := x; M[x_ /; IntegerQ[x], y_String] := Module[{a, b, c}, x]; M[x_, y_] := Module[{a, b, c}, "agn"; x + y]; M[x_String] := x; M[x_ /; ListQ[x], y_] := Block[{a, b, c}, "agn"; Length[x] + y];

In[2387]**:= Headings[M]**
Out[2387]= {{"Block", "M[x_ /; ListQ[x], y_]"}, {"Function", "M[x_, y_, z_]",

" M[x_String]"}, {"Module", "M[x_ /; x=== "avz", y_]", "M[x_ /; x== "avz"]", "M[x_ /; IntegerQ[x], y_String]", "M[x_, y_]"}} In[2388]**:= V1[x_] = x; Map[Headings, {L, L1, 80, Sin, agn, V1}]** Out[2388]= {{"Function", "L[x_]"}, {"Block", "L1[x_, y_]"}, Headings[80],

Headings[Sin] , Headings[agn], {"Function", "V1[x_]"}} In[2389]**:= G[x_] := x; Headings[G]**
Out[2389]= {"Function", "G[x_]"}
In[2390]**:= h = 80; P[x_] := Module[{a = 80, b = 480}, h = (a + b)*x; h^2];**

{**Headings[P], h**}
Out[2390]= {{"Module", "P[x_]"}, 80}

On *x* arguments different from the block/function/module, the procedure call**Headings[*x*]** is returned unevaluated. This tool is of interest, first of all, from the programmer standpoint. In a number of the appendices which use the procedural programming, the**Headings** procedure is useful enough. At that, the given procedure along with standard means uses and our means such as**BlockFuncModQ, PureDefinition** and**HeadPF** that are considered in the present book and in [28-33]. Examples of the previous fragment very visually illustrate structure of the results returned by the given procedure.

In a number of the appendices which widely use procedural programming, a rather useful is the**HeadingsPF** procedure which is an expansion of the previous procedure. Generally the procedure call**HeadingsPF[]** returns the nested list, whose elements are the sublists defining respectively headings of functions, blocks and modules whose definitions have been evaluated in the current session**;** the first element of each such sublist defines an object type in the context of {*"Block"*,*"Module"*, *"Function"*} while the others define the headings corresponding to it. The procedure call returns the*simple* list if any of sublists doesn't contain headings**;** at that, if in the current session the evaluations of definitions of objects of the*specified* three types weren't made, the procedure call returns the*empty* list. At that, the procedure call with any arguments is returned unevaluated. The fragment below represents source code of the**HeadingsPF** procedure along with examples of its typical usage.

In[2913] **:= HeadingsPF[x___ /; SameQ[x, {}]] := Module[{a = {}, d = {}, k = 1, b, c = {{"Block"}, {"Function"}, {"Module"}}, t}, Map[If[Quiet[Check[BlockFuncModQ[#], False]], AppendTo[a, #], Null] &, Names["`*"]]; b = Map[Headings[#] &, a]; While[k <= Length[b], t = b[[k]]; If[NestListQ[t], d = Join[d, t], AppendTo[d, t]]; k++]; Map[If[#[[1]] == "Block", c[[1]] = Join[c[[1]], #[[2 ;; –1]]], If[#[[1]] == "Function", c[[2]] = Join[c[[2]], #[[2 ;; –**

1]]], c[[3]] = Join[c[[3]], #[[2 ;; –1]]]]] &, d]; Select[c, Length[#] > 1&]’ If[Length[c] == 1, c[[1]], c]]

In[2914] **:= M[x_ /; SameQ[x, “avz”], y_] := Module[{a, b, c}, y]; L1[x_] := x; M[x_, y_, z_] := x + y + z; L[x_, y_] := x + y;**
M[x_ /; x == “avz”] := Module[{a, b, c}, x];
M[x_ /; IntegerQ[x], y_String] := Module[{a, b, c}, x]; M[x_, y_] := Module[{a, b, c}, “agn”; x + y]; M[x_String] := x; M[x_ /; ListQ[x], y_] := Block[{a, b, c}, “agn”; Length[x] + y];

F[x_ /; SameQ[x, “avz”], y_] := {x, y}; F[x_ /; x == “avz”] := x In[2915]:= **HeadingsPF[]**
Out[2915]= {{”Block“, “M[x_ /; ListQ[x], y_]”},

{ “Function“, “F[x_/; x=== "avz", y_]“, “F[x_/; x== "avz"]“, “L[x_, y_]“, “L1[x_]“, “M[x_, y_, z_]“, “M[x_String]”},
{”Module“, “M[x_ /; x=== "avz", y_]“, “M[x_, y_]“,
“M[x_ /; x== "avz"]“, “M[x_ /; IntegerQ[x], y_String]”}}

Reloading of the system without activation of the user means of the specified three types (Block, Function, Module)

In[2490]**:= HeadingsPF[]**
Out[2490]= {}

In[2491] **:= V[x_, y_] := v*y; F[x_String] := x <>“avz”; G[x_] := x^2; L[y_] := y**
In[2492]**:= HeadingsPF[]**
Out[2492]= {”Function“, “F[x_String]“, “G[x_]“, “L[y_]“, “V[x_, y_]”}

At that, the given procedure along with standard means uses and our tools such as**BlockFuncModQ, Headings** and**NestListQ** that are considered in the present book and in [28-33]. Examples of the previous fragment enough visually illustrate structure of the results returned by the procedure. But it must be kept in mind that performance of the procedure directly depends on stage of the current session when the**HeadingsPF** procedure has been called and how many definitions for the user means of the type {*Function*,*Module*, *Block*} were calculated in the**Mathematica** current session.

In certain problems of processing of the *headings* at times arises the question of evaluation of the*name* of a heading whose decision a very simple function gives whose call**HeadName[*x*]** returns the name of a heading***x*** in the string format provided that the heading is distinguished by procedure**HeadingQ** or**HeadingQ1** as a syntactically correct heading, i.e. the call**HeadingQ[*x*]** or **HeadingQ1[*x*]** returns*True;* otherwise, the function call**HeadName[*x*]** will be returned unevaluated. The following fragment represents source code of the**HeadName** function along with typical examples of its usage.

In[2645] **:= HeadName[x_ /; HeadingQ[x] || HeadingQ1[x]] := StringTake[x, {1, StringPosition[x, “[“, 1][[1]][[1]]–1}]**
In[2646]**:= Map[HeadName, {“V[x_/; StringQ[x]]”, “G[x_String]”, “S[x_/; IntegerQ[x]]”, “Kr[x_/; StringQ[x], y__]”, “Art[]”}]**
Out[2646]= {”V“, “G“, “S“, “Kr“, “Art”}

In[2647]**:= Map[HeadName,** {**"V[j_; StringQ[j]]", "S[j/; IntegerQ[j]]"**}**]**
Out[2647]= {HeadName["V[j_;StringQ[j]]"]**,**HeadName["S[j/;IntegerQ[j]]"]}

In some cases of procedural programming, for example, in case of necessity of insertion of calls of procedures/functions on the basis of their headings into structures of string type, the**HeadToCall** procedure is represented as a quite useful tool, whose call**HeadToCall[*h*]** in string format returns the call of a procedure/function on the basis of its heading on**`***pure`formal arguments (i.e. without the tests for an admissibility ascribed to them)***,** where***h* –** admissible heading of a procedure/function. The following fragment represents source code of the**HeadToCall** procedure along with examples of its usage.

In[2511] **:= HeadToCall[j_ /; HeadingQ[j]] := Module[**{**a = HeadName[j], b**}**, b = "**{**" <> StringTake[StringReplace[j, a <> "["–> "", 1],** {**1,–2**}**] <> "**}**"; b = Select[StrToList[b], ! StringFreeQ[#, "_"] &]; b = Map[StringTake[#,** {**1, Flatten[StringPosition[#, "_"]][[1]]–1**}**] &, b]; a <> "[" <> StringTake[ToString[b],** {**2,–2**}**] <> "]"]**

In[2512] **:= HeadToCall["G[x_, y_/; StringQ[y], z_/; MemberQ[**{**0, 1, 2**}**, z], t_Symbol, h_/; IntegerQ[h], z__, p___]"]**
Out[2512]= **"**G[x**,** y**,** z**,** t**,** h**,** z**,** p]**"**
In[2513]**:= HeadToCall["V[x_List, y_/; PrimeQ[y] && y < 90, z_/; ! HowAct[z], t_Integer, z__, p___]"]**
Out[2513]= **"**V[x**,** y**,** z**,** t**,** z**,** p]**"**

At that, it must be kept in mind that the procedure call returns also optional arguments of the studied heading.

In the light of possibility of existence in the current session of **Mathematica** of the procedures of the same name with different headings the problem of removal from the session of a procedure with concrete heading represents a certain interest**;** this problem is solved by the procedure**RemProcOnHead,** whose source code along with examples of usage are represented below.

In[2437]**:= RemProcOnHead[x_ /; HeadingQ[x] || HeadingQ1[x] || ListQ[x] && DeleteDuplicates[Map[HeadingQ[#] &, x]] ==** {**True**}**] :=**

Module[{**b, c, d, p, a = HeadName[If[ListQ[x], x[[1]], x]]**}**, If[! MemberQ[Names["`*"]||! HowAct[a], a], $Failed, b = Definition2[a]; c = b[[1 ;;–2]]; d = b[[–1]]; ToExpression["ClearAttributes[" <> a <> "," <> ToString[d] <> "]"]; y = Map[StandHead, Flatten[**{**x**}**]]; p = Select[c, ! SuffPref[#, x, 1] &]; ToExpression["Clear[" <> a <> "]"]; If[p ==** {}**, "Done", ToExpression[p]; ToExpression["SetAttributes[" <> a <> "," <> ToString[d] <> "]"]; "Done"]]]**

In[2438] **:= M[x_ /; SameQ[x, "avz"], y_] := Module[**{**a, b, c**}**, y]; L1[x_] := x; M[x_, y_, z_] := x + y + z; L[x_, y_] := x + y;**
M[x_ /; x == "avz"] := Module[{a, b, c}, x];
M[x_ /; IntegerQ[x], y_String] := Module[{a, b, c}, x]; M[x_, y_] := Module[{**a, b, c**}**, "agn"; x + y]; M[x_String] := x; M[x_ /; ListQ[x], y_] := Block[{a, b, c}, "agn"; Length[x] + y]; F[x_ /; SameQ[x, "avz"], y_] :=** {**x, y**}**; F[x_ /; x == "avz"] := x**

In[2439] **:= Definition[M]**

```
Out[2439]= M[x_ /; x=== “avz“, y_]:= Module[{a, b, c}, y]
M[x_, y_, z_]:= x+ y+ z
M[x_ /; x== “avz”]:= Module[{a, b, c}, x]
M[x_ /; IntegerQ[x], y_String]:= Module[{a, b, c}, x] M[x_ /; ListQ[x], y_]:= Block[{a,
b, c}, “agn“; Length[x]+ y] M[x_, y_]:= Module[{a, b, c}, “agn“; x+ y]
M[x_String]:= x
In[2440]:= RemProcOnHead[{“M[x_,y_,z_]”, “M[x_ /;ListQ[x],y_]”}]
Out[2440]= “Done”
In[2441]:= Definition[M]
Out[2441]= M[x_ /; x=== “avz“, y_]:= Module[{a, b, c}, y]
M[x_ /; x== “avz”]:= Module[{a, b, c}, x]
M[x_ /; IntegerQ[x], y_String]:= Module[{a, b, c}, x] M[x_, y_]:= Module[{a, b, c},
“agn“; x+ y]
M[x_String]:= x
In[2530]:= G[x_, y_ /; IntegerQ[y]] := x + y
In[2531]:= RemProcOnHead[“G[x_,y_ /;IntegerQ[y]]”]
Out[2531]= “Done”
In[2532]:= Definition[G]
Out[2532]= Null
In[2533]:= Definition[F]
F[x_ /; x=== “avz“, y_]:= {x, y}
F[x_ /; x== “avz”]:= x
In[2534]:= RemProcOnHead[“F[x_ /;x=="avz"]”]
Out[2534]= “Done”
In[2535]:= Definition[F]
Out[2535]= F[x_ /; x=== “avz“, y_]:= {x, y}
In[2541]:= V[x_] := Module[{}, x^6]; V[x_Integer] := x^2; Definition[V]
Out[2541]= V[x_Integer]:= x^2
V[x_]:= Module[{}, x^6]
In[2542]:= {RemProcOnHead[“V[x_Integer]”], RemProcOnHead[“V[x_]”]}
Out[2542]= {”Done“, “Done”}
In[2543]:= Definition[V]

Out[2543] = Null
In[2544]:= Map[RemProcOnHead, {“L[x_, y_]”, “L1[x_]”}]
Out[2544]= {”Done“, “Done”}
In[2545]:= {Definition[L1], Definition[L]}
Out[2545]= {Null, Null}

In[2508]:= StandHead[x_ /; HeadingQ[x] || HeadingQ1[x]] :=

Module[ {a = HeadName[x], b}, b = StringReplace[x, a <> “[“–> ””, 1]; b =
ToString1[ToExpression[“{” <> StringTake[b, {1,–2}] <> “}“]]; a <> “[” <>
StringTake[b, {2,–2}] <> “]”]

In[2509] := StandHead[“V[x_,y_Integer,z_/;StringQ[z]]”]
Out[2509]= “V[x_, y_Integer, z_ /; StringQ[z]]”
In[2510]:= StandHead[“F[x_/;x==="avz",y_]”]
```

```
Out[2510]= "F[x_ /; x=== "avz", y_]"
```

The successful call of the procedure **RemProcOnHead[*x*]** returns ***"Done",*** having removed from the current session a procedure/function or their list with heading or accordingly with list of the headings***x*** that are given in the string format**;** at that, on inadmissible factual argument***x*** the procedure call returns**$Failed** or is returned unevaluated. At the same time, the remaining subobjects with the name of the object***x*** which have been processed by the procedure**RemProcOnHead** save and options, and attributes, except in the case when the object***x*** is removed completely. At that, it is necessary to do ***2*** remarks, namely**:*(1)*** the means of this fragment that are used as examples have only formally correct code, no more, and***(2)*** heading in the procedure call**RemProcOnHead[*x*]** is coded according to the format**ToString1[*x*].**

The previous fragment contains source code of the **RemProcOnHead** with examples of its usage along with control of the obtained results. It must be kept in mind that realization of algorithms of a number of the procedures that significantly use the headings requires the coding of headings in the format corresponding to the system agreements at evaluation of definitions of a procedure/function/block. For automation of representation of a*heading* in the standard format the**StandHead** procedure can be quite useful whose source code along with examples of its application completes the previous fragment. The procedure call**StandHead[*h*]** returns the heading of a block/ procedure/function in the format corresponding to the*system* agreements at evaluation of its definition.

So, if in the **Maple** the identifier of a procedure/function is its name, then in the**Mathematica** system this function is carried out by its heading, i.e. the construction of kind***"Name***[*List of formal arguments*]**"** that it is necessary to consider at programming of means for processing of the specified objects. Therefore the**Names** function needs to be applied in combination with the **Definition** function because the first returns only the names of procedures and functions and tells nothing about existence in the current session of the user procedures/functions of the same name with different headings as nice illustrates the following simple fragment, namely**:**

```
In[2620] := G[x_] := Module[{a = 90}, x^2 + a];
G[x_ /; PrimeQ[x]] := Module[{a = 90}, x + a];
V[x_ /; ListQ[x]] := Module[{}, Length[x]];
V[x_] := Module[{}, x^2]

In[2621] := Select[Names["`*"], ProcQ1[#] &]
Out[2621]= {"G", "V"}
In[2622]:= Definition[G]

Out[2622] = G[x_ /; PrimeQ[x]]:= Module[{a= 90}, x+ a]
G[x_]:= Module[{a= 90}, x^2+ a]
In[2623]:= Definition[V]
Out[2623]= V[x_ /; ListQ[x]]:= Module[{}, Length[x]]
V[x_]:= Module[{}, x^2]

In[2624]:= MdP[x___] := Module[{b = {}, c, d,
a = Select[Names["`*"], BlockFuncModQ[#] &]}, d = Flatten[Map[ToString, {x}]];

a = If[a == {}, {}, If[d == {}, a, If[MemberQ4[a, d], Intersection[a, d], {}]]]; If[a ==
```

{}, $Failed, c = Map[AppendTo[b, {#, Length[Flatten[{PureDefinition[#]}]]}] &, a] [[-1]]; If[Length[c] > 1, c, c[[1]]]]]

```
In[2625]:= MdP[]
Out[2625]= {{"G", 2}, {"V", 2}}

In[2626] := MdP[G, V, H]
Out[2626]= {{"G", 2}, {"V", 2}}
In[2627]:= Clear[G, V]; MdP[]
Out[2627]= $Failed
In[2628]:= MdP[G1, V1, H]
Out[2628]= $Failed
In[3502]:= MdP[]
Out[3502]= {{"F", 2}, {"G", 2}, {"L", 1}, {"L1", 1}, {"M", 7}}
```

The previous fragment defines the procedure, whose call **MdP[*x*]** returns a simple***2***–element list, in which the first element– an object name in string format and the second element– number of headings with such name*(if**x** defines a procedure/function/block activated in the current session;in the absence of similar object$Failed is returned);* the nested list whose***2***–element sublists have the structure described above*(if an object**x**defines the list of the modules/ functions/blocks activated in the current session)*, the nested list of the previous format*(if**x**is empty,defining the list of all functions/blocks/modules activated in the current session);* in the absence of the functions/modules/blocks activated in the current session the call**MdP** returns**$Failed.** At that, the procedure along with standard means uses and our means such as**BlockFuncModQ, PureDefinition** and**MemberQ4** which are considered in the present book and in [28-33]. Examples of the previous fragment rather visually illustrate structure of the results returned by the given procedure.

As it was already noted, the current session may contain several different definitions of a procedure/function/block with the same name which differ only at the level of their headings. The procedure call**Definition2[*x*]** in an *optimum* format returns the list of all definitions in string format of a block/ procedure/function with a name***x*,** accompanying it with options and the list of the attributes that are ascribed to a symbol***x*.** According to the system agreements the procedures/functions/blocks of the*same name* have the same ascribed options and attributes as illustrates the following fragment:

```
In[3389] := G[x_, y_] := x^2*y^2
In[3390]:= Options[G] = {Art–> 25, Kr–> 18};
In[3391]:= SetOptions[G, Art–> 25, Kr–> 18]
Out[3391]= {Art–> 25, Kr–> 18}

In[3392] := Definition2[G]
Out[3392]= {"G[x_, y_]:= x^2*y^2", "Options[G]:= {Art–> 25, Kr–> 18}", {}}
In[3393]:= G[x_] := x^2; G[x_, y_, z_] := x + y + z;

SetAttributes[G, {Protected, Listable}]
In[3394]:= Definition2[G]
Out[3394]= {"G[x_, y_]:= x^2*y^2", "G[x_]:= x^2", "G[x_, y_, z_]:= x+y+z",

"Options[G]:= {Art–> 25, Kr–> 18}", {Listable, Protected}} In[3395]:= DefOnHead[x_
```

/; HeadingQ[x]] := Module[{a, b, c, d, h = RedSymbStr[StringReplace[StandHead[x], ",”–> ", "], " ", " "]}, a = HeadName[h]; b = Definition2[ToExpression[a]];

c = Select[b, SuffPref[#, Map3[StringJoin, h, {" := ", " = "}], 1] &]; d = Select[b, SuffPref[#, Quiet[Map3[StringJoin, "Options[" <> a <> "]", {" = ", " := "}]], 1] &]; If[MemberQ[b, "Undefined"], $Failed,

If[d == {}, AppendTo[c, b[[–1]]], Join[c, {d[[1]], b[[–1]]}]]]]] In[3396]:= **DefOnHead["G[x_,y_,z_]"]**
Out[3396]= {"G[x_, y_, z_]:= x+ y+ z", "Options[G]:= {Art–> 24, Kr–> 17}",

{Listable , Protected}}
In[3397]:= **DefOnHead["G[x_,y_]"]**
Out[3397]= {"G[x_, y_]:= x^2*y^2", "Options[G]:= {Art–> 24, Kr–> 17}",

{Listable , Protected}}
In[3398]:= **DefOnHead["G[x_]"]**
Out[3398]= {"G[x_]:= x^2", "Options[G]:= {Art–> 24, Kr–> 17}",

{Listable, Protected}}

For receiving of definition of a procedure/function/block ***x*** with the given heading*(the main identifier)* a number of means one of which is presented by the previous fragment is created**;**more precisely the**DefOnHead** procedure whose call**DefOnHead[*j*]** returns the list whose the*first* element– definition in string format of a procedure/function/block with the given heading***j****(or the list of definitions for the subobjects of the same name)* whereas other elements are options*(if they are)* and list of attributes ascribed to the function/block/ procedure***x*.** At that, the following defining relation**HeadName[*j*] =*x*** takes place. As a whole, it is recommended to use a certain unique name for each definition, for providing of such possibility the system functions**Clear** and **ClearAll** can be used at modifications of means if their headings change. Thus, at the call of a procedure/function/block from the list of definitions of its subobjects a definition with the heading corresponding to the actual arguments, i.e. that are admissible for formal arguments with the ascribed tests for an admissibility is chozen. Moreover, a heading of the format**G[*x*_, *y*_, *z*_, …]** has the minimum priority among the headings of other formats irrespective of the evaluation order in the current session of definitions of procedures/functions/blocks of the same name as very visually illustrates the following rather simple fragment, namely:

In[2863] **:= G[x_, y_] := StringJoin[x, y] <> "RansIan"**
In[2864]:= **G[x_Integer, y_Integer] := x + y**
In[2865]:= **G[x_String, y_Integer] := y*StringLength[x]**
In[2866]:= **Definition2[G]**
Out[2866]= {"G[x_Integer, y_Integer]:= x+ y", "G[x_String, y_Integer]:=

y *StringLength[x]",
"G[x_, y_]:= StringJoin[StringJoin[x, y], "RansIan"]", {}} In[2867]:= {**G[80, 90], G["AvzAgnVsvArtKr", 500]**}
Out[2867]= {170, 7000}
In[2868]:= **G["AvzAgnVsvArtKr", "Tallinn"]**
Out[2868]= "AvzAgnVsvArtKrTallinnRansIan"
In[2869]:= **G[x_, y___] := If[{y}== {}, x^2, {y}= {x}; x^2]; G[500]** Out[2869]= 250

000
In[2870]:= **G[“90”, “500”]**
Out[2870]= “90500RansIan”
In[2871]:= **ClearAll[G]**
In[2872]:= **G[x_] := x^2; G[x_, y_ /; ! HowAct[y] === Null] := {y = x, x^2}[[2]]**
In[2873]:= **Definition2[G]**
Out[2873]= {”G[x_]:= x^2“, “G[x_, y_ /; !HowAct[y]=== Null]:=

{y= x, x^2}[[2]]“, {}}

Above, it was already noted that in the most cases is expedient to use only one definition of a procedure/function/block, that at times quite significantly simplifies its processing. Meanwhile, in certain cases is quite convenient the usage of a number of the procedures/functions/blocks of the same name, for example, for the purpose of simplification of their program realization. So, realization of the**G** function from undefinite number of formal arguments of the***2nd*** part of the previous fragment can serve as an example. Definition of two**G** functions covering all cases of the function**G** in some cases allows to simplify realization. In this example such simplification isn’t so obvious since it only illustrates reception while in case of rather complex procedures which in the body have to execute processing of undefinite quantity of the received actual arguments such approach can be very effective.

As it was noted above, generally the user procedure /function/block can has both the ascribed attributes, and options. At that, some of earlier considered means were based, mainly on the call of our**Definition2[*x*]** of our procedure returning the list whose last element contains the list of*attributes* ascribed to symbol***x*** whereas the sublist of**Definition2[*x*][[*1*;;–*2*]]** contains definitions of a procedure/function/block together with options if those exist. The next **PureDefinition** procedure solves the problem of receiving of*pure definitions* of a procedure/function/block without options and the ascribed attributes.

In[2826]:= **G[x_] := x; G[x_, y_ /; ! HowAct[y] === Null] := {y = x, x^2}[[2]]**
In[2827]:= **Options[G] = {Art–> 25, Kr–> 18};**

SetOptions[G, Art –> 25, Kr–> 18]
Out[2827]= {Art–> 25, Kr–> 18}
In[2828]:= **SetAttributes[G, {Listable, Protected}]; Definition2[G]** Out[2828]= {”G[x_]:=x“, “G[x_, y_ /; !HowAct[y]=== Null]:= {y= x, x}[[2]]“,

“Options[G]:= {Art–> 25, Kr–> 18}“, {Listable, Protected}} In[2834]:=
PureDefinition[x_, t___] := Module[{b, c, d,
h = ToString[x] <> ” /: Default[“, a = If[UnevaluatedQ[Definition2, x], $Failed, Definition2[x]]},

If[a === $Failed, Return[$Failed]]; b = a[[1 ;; –2]]; c = If[SuffPref[b[[–1]], Map3[StringJoin, “Options[” <> ToString[x] <> “]”, {” = “, ” := “}], 1], b[[1 ;;–2]], b]; If[{t}!= {}&& ! HowAct[t], d = MinusList[a, c]; Join[If[Length[d] > 1, d, Flatten[d]], Select[a, SuffPref[#, h, 1] &]]]; c = Select[c, ! SuffPref[#, h, 1] &]; If[Length[c] == 1, c[[1]], c]]

In[2835]:= {**PureDefinition[G, t], t**}

Out[2835]= {{"G[x_]:=x", "G[x_, y_ /; !HowAct[y]===Null]:= {y=x,x}[[2]]"},

{ "Options[G]:= {Art–> 25, Kr–> 18}", {}}}
The procedure call**PureDefinition[*x*]** returns definition in string format or their list of a block/function/module***x*** without options, ascribed attributes and values by default for formal arguments while the call**PureDefinition[*x, t*]** with the second optional argument***t*** **–** an undefinite variable**–** through it returns the list of the options, attributes and values by default attributed to symbol***x.*** In the case of inadmissible argument***x*** the procedure call returns **$Failed,** including also a call on the**Compile** functions. The fragment above represents source code of the**PureDefinition** procedure with examples of its application. The**PureDefinition** procedure is represented as useful tool in various processings of definitions of blocks/functions/modules. Procedure is rather widely used in series of means of our package***AVZ_Package*** [48].

The concept of the **Mathematica** allows existence of a few blocks, functions or procedures, of the same name that are identified by their*headings* but not names. Operating with these objects is supported by a number of the means presented in the given book and in the package***AVZ_Package*** [48]. In this connection the procedure**ExtrProcFunc** represents a certain interest, whose call**ExtrProcFunc[*h*]** returns an unique name of a generated block/function/ procedure that in the list of definitions has a heading***h;*** otherwise,**$Failed** is returned. The procedure is characteristic in that leaves all definitions of a symbol**HeadName[*h*]** without change. At that, the returned object saves all options and attributes ascribed to the symbol**HeadName[*h*].** The following fragment represents source code of the**ExtrProcFunc** procedure along with the most typical examples of its application.

In[2860] **:= ExtrProcFunc[x_ /; HeadingQ[x]] := Module[{a = StandHead[x], c, d, b = HeadName[x], g, p}, c = Definition2[ToExpression[b]]; If[c[[1]] == "Undefined", $Failed, d = Select[c, SuffPref[#, a <> " := ", 1] &]; c = ToString[Unique[b]]; If[d != {}, ToExpression[c <> d[[1]]]; g = AttrOpts[b]; p = c <> b; Options[p] = g[[1]]; SetOptions[p, g[[1]]]; ToExpression["SetAttributes[" <> p <> "," <> ToString[g[[2]]] <> "]"]; Clear[c]; p, Clear[c]; $Failed]]]**

In[2861]**:= H[x_] := x^2; H[x_, y_] := x + y; H[x_, y_, z_] := x + y + x; H[x_Integer] := x; H[x_, y_Integer] := x + y; H[x_String] := x <> "Agn"**

Options[H] = {Art–> 25, Kr–> 18}; SetOptions[H, {Art–> 25, Kr–> 18}]; SetAttributes[H, {Listable, Protected}] In[2862]**:= Definition2[H]**

Out[2862] = {"H[x_Integer]:= x", "H[x_String]:= StringJoin[x, "Agn"]", "H[x_]:= x^2", "H[x_, y_Integer]:= x+ y", "H[x_, y_]:= x+ y", "H[x_, y_, z_]:= x+ y+ x", "Options[H11H]= {Art–> 25, Kr–> 18}", {Listable, Protected}}

In[2863] **:= ExtrProcFunc["H[x_,y_,z_]"]**
Out[2863]= "H11H"
In[2864]**:= Definition["H11H"]**
Out[2864]= Attributes[H11H]= {Listable, Protected}

H11H[x _, y_, z_]:= x+ y+ x
Options[H11H]= {Art–> 25, Kr–> 18}
In[2865]**:= ExtrProcFunc["H[x_,y_,z_String]"]**

Out[2865]= **$**Failed
In[2866]**:= ExtrProcFunc[“H[x_String]”]**
Out[2866]= **“**H12H**”**
In[2867]**:= Definition[“H12H”]**
Out[2867]= Attributes[H12H]= {Listable**,** Protected}
H12H[x_String]**:=** x<> **“**Agn**”**
Options[H12H]= {Art–> 25**,** Kr–> 18}
In[2868]**:= H12H[“AvzAgnVsvArtKr”]**
Out[2868]= **“**AvzAgnVsvArtKrAgn**”**
In[2869]**:= H11H[42, 2014, 72]**
Out[2869]= 2098

In[3543]**:= AttrOpts[x_ /; BlockFuncModQ[x]] := Module[**{**b, c, d, a = Definition2[x]**}**, b = a[[–1]]; c = Select[a, SuffPref[#,“Options[” <> ToString[x] <> “]”, 1] &]; If[c == **{}**, d = c, d = StringSplit[c[[1]], ” := “][[2]]]; **{**ToExpression[d], b**}**]**

In[3544] **:= AttrOpts[H]**
Out[3544]= {{Art–> 25**,** Kr–> 18}**,** {Listable**,** Protected}}
In[3545]**:= Sv[x_, y_] := x^2 + y^2; AttrOpts[Sv]**
Out[3545]= {{}**,** {}}
At that, the procedure along with standard means uses and our means such as**HeadingQ, Definition2, HeadName, StandHand, SuffPref** and**AttrOpts** which are considered in the present book and in [28-33]. Moreover, the last **AttrOpts** procedure completes the previous fragment. The procedure call **AttrOpts[*x*]** returns the***2***-element nested list whose*first* element determines options whereas the*second* element defines the list of the attributes ascribed to a symbol***x*** of type {*Block***,***Funcion***,***Module*}. On a symbol***x*** without options and attributes ascribed to it, the call**AttrOpts[*x*]** returns {{}**,** {}}. Examples of the previous fragment rather visually illustrate structures of the results that are returned by both the procedure**ExtrProcFunc,** and the**AttrOpts.**

At that, in definition of the **ExtrProcFunc** procedure one artificial reception essential in practical programming has been used. So, direct application of our and standard means {**Attributes, ClearAllAttributes, SetAttributes**} for processing of attributes in body of a procedure in certain cases doesn**’**t give of the desired result therefore it is necessary to use special constructions the organization of which is rather transparent and doesn**’**t demand any special explanations. The reception represented in source code of the**ExtrProcFunc** procedure from the previous fragment is used in some other means that are represented in the present book and in our package***AVZ_Package*** [48]. So, the procedure**RemProcOnHead,** considered above, also significantly uses the given reception.

The call **ActBFM[]** of the next rather simple function returns the list of*names* in string format of the user blocks, functions and modules, whose definitions have been activated in the current**Mathematica** session. The next fragment represents source code of the function along with an example of its usage.

In[2824] **:= ActBFM[] := Select[Names[“Global`*”], ! TemporaryQ[#] && BlockFuncModQ[#] &]**
In[2825]**:= ActBFM[]**

Out[2825]= {"ActBFM", "Agn", "Avz", "B", "f", "F", "M", "Name", "RansIan"}
The above**ActBFM** function has a number of interesting enough appendices at programming of various problems, first of all, of the system character. In particular, the**ActBFM** function plays a rather essential part at search of the user objects, whose definitions have been evaluated in the current session of the**Mathematica** system.

6.5. Formal arguments of procedures and functions; the means of processing them in the*Mathematica*software

Having considered in the previous two sections the means of manipulation with definitions of blocks/functions/modules, and also their headings, we pass to consideration of means whose scope of interests includes a number of important problems connected with manipulation by formal arguments that compose headings of definitions of the user procedures and functions. At that, these components are extremely important and their total absence in headings doesn't allow system in general to consider objects with similar headings as procedures or functions. In the previous section the means of processing of*headings* of procedures/blocks/functions have been considered from which procedure**HeadingQ1** in the best way tests an arbitrary string as a heading what very visually illustrates the following simple example: In[2546]:= **Map[HeadingQ1, {"G[]", "G[ ]", "G[ ]"}]**
Out[2546]= {False, False, False}
In[2547]:= **G[] := x; {FunctionQ[G], Clear[G]}[[1]]**
Out[2547]= False
In[2548]:= **G[x_ /; SameQ[{x}, {}]] := x; FunctionQ[G]**
Out[2548]= True
In[2549]:= **HeadingQ["G[x_ /; SameQ[{x}, {}]]"]**
Out[2549]= True
In[2550]:= **G[x___] := {x}; G[]**
Out[2550]= {}
In[2551]:= {**HeadingQ["G[x___]"], HeadingQ1["G[x___]"]**}
Out[2551]= {True, True}
Of the represented example follows, that strings of the type**"G[ ]"** can't be considered as syntactic correct headings, and definitions on their basis can't be considered as procedures or functions. Meanwhile, in case of necessity to define procedures or functions whose calls make sense on the empty list of actual arguments, it is possible to code their*headings* as it is stated above; in this case our means identify them properly. Further consideration of means of manipulation with formal arguments of procedures, blocks and functions assumes short introduction into templates concept*;* in more detail the given question is considered in help on the system and, in particular, in book [33]. ***Templates(patterns)*** are used in the**Mathematica** for representation of the classes of expressions. Very simple example of a template is an expression ***h*[*x_*]** that represents a class of expressions of type***h*[*any expression*].** As the prerequisite of introduction of the concept*"Template"* into the**Mathematica** the fact served, what many enough operations support work not only with separate expressions but also with templates representing the whole classes of expressions. So, in particular, it is possible to use the templates in rules of transformation

for the indicating of that how properly to transform classes of expressions. The templates can be used for calculation of positions of all expressions in some certain class along with a number of other applications of the sufficiently developed templates mechanism.

The basic identifier that defines, practically, all templates in **Mathematica** is the**"_"** symbol*(symbol of underlining)* that is being ascribed to some symbol on the right. In this case the**Mathematica** system considers such symbol as any admissible expression used as its value. The call**Head[*x*]** of the earlier mentioned function on a pattern***x*** returns**Pattern** while the call**PatternQ[*x*]** of very simple function returns*True* if***x* –** a template, and*False* otherwise**:**

```
In[2570]:= PatternQ[x_] := If[Head[x] === Pattern, True, False] In[2571]:=
Map18[{PatternQ, Head}, {agn_, _, _a _, x_, _^_, avz___, __}] Out[2571]= {{True,
False, False, True, False, True, False},
{Pattern, Blank, Times, Pattern, Power,Pattern, BlankSequence}} In[2572]:= Map18[x_
/; ListQ[x] && DeleteDuplicates[Map[SymbolQ[#] &, x]] == {True}, y_ /; ListQ[y]]
:= Map[Map[#, y] &, x] In[2573]:= Map18[{X, Y, Z}, {a, b, c}]
Out[2573]= {{X[a], X[b], X[c]}, {Y[a], Y[b], Y[c]}, {Z[a], Z[b], Z[c]}}
```

Along with the **PatternQ** function and comparative examples for it and the standard**Head** function the previous fragment represents quite simple and useful**Map18** function in many appendices in addition to the represented means of the***Map***–group. The call**Map18[*x, y*],** where***x* –** the list {***x1, x2, …, xn***} of symbols and***y* –** the list {***y1, y2, …, yp***} of any expressions, returns the nested list of the following format, namely**:**

{{***x1*[*y1*],*x1*[*y2*], …,*x1*[*yp*]**}, {***x2*[*y1*],*x2*[*y2*],…,*x2*[*yp*]**}, **…,** {***xn*[*y1*],*xn*[*y2*], …,*xn*[*yp*]**}}

The result returned by the function call **Map18[*x, y*]** is transparent enough and of ant special explanations doesn**'**t demand. In principle, it is possible to place the**"_"** symbol in any place of an expression, defining thus the pattern corresponding to some group of the expressions received by replacement of this symbol by any expression. Several simple enough examples of patterns are given below, namely**:**

h* [*x*_]**–*heading of a block/function/procedureh***with one formal argument***x***where **x** –an arbitrary expression;*
h*[*x*_,*y*_]**–*heading of a block/function/procedureh***with two formal arguments***x, y***where***x,y*** –arbitrary expressions;*
h*[*x*_,*x*_]**–*heading of a block/function/procedureh***with two identical arguments* ***x***where***x*** –an arbitrary expression;*
x*^*n*_**–*defines an arbitrary expressionx***in an arbitrary degree***n;**
x*_^*n*_**–*defines an arbitrary expressionx***in an arbitrary degree***n;**
x*_ +*y*_ +*z*_**–*определяет сумму трех произвольных выраженийx, y***и***z;**
{***x*_,*y*_,*z*_**}–*determines the list of three arbitrary expressions***x, y***and***z; *90x*_^*y*_ +500**
***x*_*y*_ +*z*_**–*defines an expression with five patterns.*

Basic patterns in the**Mathematica** are the following three patterns, namely**: _***or***Blank[]***(in the full form)–the pattern defining an arbitrary expression;*

_ *t**or***Blank[*t*]***(in the full form)–the pattern defining an arbitrary expression with a heading***t;**
__*(2symbols***"_"***) or***BlankSequence[]***(in the full form)–the pattern defining an arbitrary*

expression or sequence of arbitrary expressions;
__*t* *or* **BlankSequence[*t*]** *(in the full form)–the pattern determining an arbitrary expression or sequence of arbitrary expressions with a heading* ***h*** *each;* **___** *(***3** *symbols* **"_"** *) or* **BlankNullSequence[]** *(in the full form)–the pattern that determines absence of expressions, or sequence of arbitrary expressions;* **___*t*** *or* **BlankNullSequence[*t*]** *(in the full form)–the pattern which determines absence of expressions, or sequence of arbitrary expressions with heading* ***t*** *each.*

At that, in the full form the expressions containing patterns of types { **"___", "__", "_"**} are represented in the formats illustrated by the next example: In[2500]:= **Map[FullForm,** {**x_, x__, x___**}**]**
Out[2500]= {Pattern[x, Blank[]], Pattern[x, BlankSequence[]],

Pattern[x , BlankNullSequence[]]}
The simple enough **ExprPatternQ** function provides testing of an expression regarding existence in it of patterns of types {**"_", "__", "___"**}, whose the call **ExprPatternQ[*x*]** returns *True* if an expression ***x*** contains at least one of the patterns {**"_", "__", "___"**}, and *False* otherwise. The next fragment represents source code of the **ExprPatternQ** function with typical examples of its use:

In[2502]:= **ExprPatternQ[x_] := ! StringFreeQ[ToString[FullForm[x]],** {**"BlankSequence[]", "BlankNullSequence[]", "Blank[]"**}**]** In[2503]:= **Map[ExprPatternQ,** {**a*Sin[x], 6 x_^y_+a x_*y_, x_^y_, x__, z___**}**]** Out[2503]= {False, True, True, True, True}

The user has possibility to create patterns for expressions with an arbitrary structure, however the most widespread way of use of templates is a block/ function/procedure definition when formal arguments are specified in its *heading*. At that, the coding of formal arguments without the above patterns doesn't allow to consider these objects as the blocks/functions/procedures as illustrates a simple enough example:

In[2589] := **G[x, y] := x^2 + y^2; G1[x_, y_] := x^2 + y^2**
In[2590]:= {**G[90, 500], G1[90, 500]**}
Out[2590]= {G[90, 500], 258100}

Once again it is necessary to emphasize that patterns in the ***Math*** language represent classes of expressions with the given structure when one pattern corresponds to a certain expression and if the structure of pattern coincides with structure of an expression, i.e. by a filling of the patterns it is possible to receive an expression. Moreover, even two expressions, mathematically equivalent, can't be presented by the same template if they don't have the same structure. For example, expression ***(a + b)^2*** is equivalent to expression ***a^2+2*a*b + b^2*** however these expressions aren't equivalent at the level of patterns representing them, for the reason, that both have various full form as illustrates a simple example:

In[2507]:= **FullForm[(a + b)^2]**
Out[2507]//FullForm=

Power[Plus[a , b], 2]
In[2508]:= **FullForm[a^2 + 2*a*b + b^2]**
Out[2508]//FullForm=

Plus[Power[a , 2], Times[2, a, b], Power[b, 2]]
The fact that patterns define structure of expressions, is very important for the solution of the problem of determination of the transformation rules of changing of structure of expressions without change of their mathematical equivalence. The system has not other general criterion which would allow to define equivalence of two expressions. For realization of algorithm of the *comparison* of expressions the system uses reduction them upto the full form determined by the**FullForm** function. In the reference on the**Mathematica** a number of important mechanisms of creation of patterns for a quite wide class of expressions is discussed while in other manuals the receptions used by the**Mathematica** for the purpose of expansion and restriction of classes of expressions represented by patterns are being considered. For definition of the expressions coinciding with the given pattern it is possible to apply the**Cases** function allowing five coding formats [68]; so, the call**Cases[*a,p*]** according to the first format returns the elements-expressions of a list***a*** that are structurally corresponded to a pattern***p*** as very visually illustrates the following simple example, namely:
In[2610]:= **Cases[{a+b*c^5, 5+6*y^7, a+b*p^m, a+b*m^(–p)}, a+b*x_^n_]**
Out[2610]= {a+ b c^5, a+ b p^m, a+ b m^–p}

Meanwhile, without being distracted by details, we only will note that the **Mathematica** has a number of the functions providing the functioning with expressions at the level of the patterns representing them as in general, and at the level of the subexpressions composing them; furthermore, the reader can familiarize oneself with these means, in particular, in [51,60,65,68,71].

As it was noted, it is possible to apply the **Cases** function for determination of the expressions coinciding with a given*pattern,* however not all problems of expressions comparison with patterns are solved by the standard means. For solution of the problem in*broader* aspect the**EquExprPatt** procedure can be rather useful whose call**EquExprPatt[*x,p*]** returns*True* if an expression***x*** corresponds to a given pattern***p,*** and*False* otherwise. The fragment below represents source code of the procedure with examples of its application.

In[3395] **:= EquExprPatt[x_, y_ /; ExprPatternQ[y]] := Module[{c, d={}, j, t, v ={}, k = 1, p, g ={}, s ={}, a =Map[FullForm, Map[Expand, {x, y}]], b = Mapp[MinusList, Map[OP, Map[Expand, {x, y}]], {FullForm}], z = SetAttributes[ToString, Listable], w}, {b, c}= ToString[{b, a}]; p = StringPosition[c[[2]], {“Pattern[“, “Blank[]”}];**

While[k = 2*k–1; k <= Length[p], AppendTo[d, StringTake[c[[2]], {p[[k]][[1]], p[[k + 1]][[2]]}]]; k++]; {t, k}= {ToExpression[d], 1}; While[k <= Length[t], AppendTo[v, StringJoin[ToString[Op[t[[k]]]]]]; k++]; v = ToString[v]; v = Map13[Rule, {d, v}]; v = StringReplace[c[[2]], v]; b =Quiet[Mapp[Select, b, ! SystemQ[#] ||

BlockFuncModQ[ToString[#]] &]]; {b, k, j}= {ToString[b], 1, 1}; While[k <= Length[b[[1]]], z = b[[1]][[k]]; AppendTo[g, {“[” <> z <> “,”–> “[w”, ” ” <> z <> “,”–> ” w”, “[” <> z <> “]”–> “[w]”, ” ” <> z <> “]”–> ” w]”}]; k++]; While[j <= Length[b[[2]]], z = b[[2]][[j]]; AppendTo[s, {“[” <> z <> “,”–> “[w”, ” ” <> z <> “,”–> ” w”, “[” <> z <> “]”–> “[w]”, ” ” <> z <> “]”–> ” w]”}]; j++]; ClearAttributes[ToString, Listable]; z = Map9[StringReplace, {c[[1]], v},

Map[Flatten, {g, s}]]; SameQ[z[[1]], StringReplace[z[[2]], Join[GenRules[Flatten[Map[# <> "," &, Map[ToString, t]]], "w"], GenRules[Flatten[Map[# <> "]" &, Map[ToString, t]]], "w]"],

GenRules[Flatten[Map[# <> ")" &, Map[ToString, t]]], "w)"]]]]]

In[3396] **:= EquExprPatt[a*Sin[x]–5*b*c^5, a*Sin[x]–5*b*x_^n_]** Out[3396]= True
In[3397]**:= EquExprPatt[a*Sin[x]–5*b*c^5, 90*Sin[x]–500*b*x_^n_]** Out[3397]= True
In[3398]**:= EquExprPatt[a^2 + 2*a*b + b^2, (x_ + y_)^2]**
Out[3398]= True
In[3399]**:= Mapp[EquExprPatt,** {**a + b*c^5, 5 + 6*y^7, a + b*p^m,**

a + b*m^p }**, a + b*x_^n_]**
Out[3399]= {True**,** True**,** True**,** True}
In[3400]**:= Mapp[Cases,** {{**a + b*c^5**}**,** {**5 + 6*y^7**}**,** {**a + b*p^m**}**,**

{**a + b*m^p**}}**, a + b*x_^n_]**

Out[3400] = {{a+ b c^5}, {}, {a+ b p^m}, {a+ b m^p}}
In[3401]**:= EquExprPatt1[a^2 + 2*a*b + b^2, (a + b)^2]**
Out[3401]= True

At that, the definition of the **EquExprPatt** along with standard means uses and our means such as**ExprPatternQ,Map9, Map13, Mapp, MinusList, Op, OP, ProcQ, QFunction, SystemQ,** which are considered in the present book and in [28-33]. The last examples of the fragment illustrate as well the more ample opportunities of the**EquExprPatt** procedure concerning the standard **Cases** function. As the algorithm of the procedure is based on presentation of expressions and patterns in the full form***(FullForm),*** in principle, as the second argument of the**EquExprPatt** procedure it is possible to apply any expression, having encoded the second argument as***y_*** in definition of the **EquExprPatt,** having modified it in the **EquExprPatt1** procedure different from the**EquExprPatt** only by this condition. In this case it is possible to test two any expressions regarding their*structural* equivalence what represents a quite important question in a number of tasks of the expressions analysis. Note, in realization of the procedure a quite useful reception of temporary endowing of the system**ToString** function by***Listable***–attribute is used. In [33] the questions of manipulations with patterns are considered in detail.

Determination of types of expression in patterns. For this purpose it is quite possible to use headings of expressions***w****(they are determined by the function call***Head[*w*]**), which define their main essence. So, the patterns***_h*** and***x_h*** will represent expressions with a heading***h,*** the next headings from which are the most often used, namely**:**

x_h – an expression x with heading h:
x_Integer –*an expression****x****with heading****Integer****(integer)*
x_Real –*an expression****x****with heading****Real****(real number)*

x _Complex –*an expression****x****with heading****Complex****(complex number)* ***x_List –****an expression****x****with heading****List****(list)*
x_String –*an expression****x****with heading****String****(string)*
x_Symbol –*an expression****x****with heading****Symbol****(symbol)*

***x_Plus** –an expression**x**with heading**Plus**(addition,subtraction) **x_Times** –an expression**x**with heading**Times**(product,division) **x_Power** –an expression**x**with heading**Power**(power)*

In principle, any admissible heading can act as some heading as a part of a pattern. We will give examples of such patterns, namely:

In[2415] **:= G[x_Plus] := x^2; S[x_Power] := x^2; {G[90], G[a + b], S[500], S[a^b], G[c–d], 5^(–1)}**
Out[2415]= {G[90], (a+ b)^2, S[500], a^(2 b), (c– d)^2, 1/5}

Meanwhile, in certain cases of standardly defined headings isn ‘t enough for assignment of patterns, quite naturally bringing up the question of addition to their list of the headings determined by the user. Since for evaluation of a heading, the standard**Head** function is used, therefore naturally to modify this function regarding testing by it of wider class of the headings. For this purpose the**RepStandFunc** procedure has been determined, whose the call **RepStandFunc[*x*,*y*, *z*]** returns the call of a function***y*** of the same name with a standard function***y*** and whose definition is given in string format by the argument***x*,** on the list***z*** of its actual arguments. At the same time such call of the**RepStandFunc** procedure is once–only in the sense that after the call the initial state of a standard function***y*** is restored. The following fragment presents source code of the**RepStandFunc** procedure and examples of its application and of testing of aftereffect of result of its call; along with that, in other part of the fragment the means illustrating the aforesaid are presented.

In[3380] **:= RepStandFunc[x_/; StringQ[x], y_/; SymbolQ[y], z_/; ListQ[x]] := Module[{c, d, b = Attributes[y], a = ToString[y] <> ".mx"}, DumpSave[a, y]; ClearAllAttributes[y]; Clear[y]; ToExpression[x]; d = y[Sequences[z]]; Clear[y]; Get[a]; SetAttributes[y, b]; DeleteFile[a]; d]**

In[3381] **:= x = "Sin[x_, y_, z_] := x^2 + y^2 + z^2";**
RepStandFunc[x, Sin, {73, 90, 500}]
Out[3381]= 263 429
In[3382]**:= x = "Sin[x_] := x^5"; RepStandFunc[x, Sin, {47}]**
Out[3382]= 229 345007
In[3383]**:= Definition[Sin]**
Out[3383]= Attributes[Sin]= {Listable, NumericFunction, Protected}
In[3384]**:= Sin[73.50090]**
Out[3384]=–0.947162

In[3390] **:= Headd := "Head[x_] := Module[{b = {ListListQ, ProcQ, SystemQ, NestListQ, QFunction}, c = {ListList, Procedure, System, NestList, Function}, h = SetAttributes[SetAttributes, Listable], d = 90, k = 1}, SetAttributes1[c, Protected]; Quiet[For[k = 1, k <= Length[b], k++, If[b[[k]][x], d = c[[k]]; Break[]]]]; ClearAttributes[SetAttributes, Listable]; If[d === 90, x[[0]], d]]"**

In[3391] **:= RepStandFunc[Headd, Head, {{{a}, {b, c}, {d}}}]**
Out[3391]= NestList
In[3392]**:= Definition[Head]**
Out[3392]= Attributes[Head]= {Protected}
In[3393]**:= Head[{{a}, {b, c}, {d}}]**

```
Out[3393]= List
In[3394]:= G[h_NestList] := Length[h]
In[3395]:= G[{{a}, {b}, {c}, {d, t}, {f}, {g}, {v}}]
Out[3395]= G[{{a}, {b}, {c}, {d, t}, {f}, {g}, {v}}]
In[3396]:= G[h_List] := Length[h]
In[3397]:= G[{{a}, {b}, {c}, {d, t}, {f}, {g}, {v}}]
Out[3397]= 7
In[3398]:= ClearAllAttributes[Head]; Clear[Head]; ToExpression[Headd] In[3399]:=
G[h_ListList] := Length[h]
In[3400]:= G[{{a}, {b}, {c}, {d, t}, {f}, {g}, {v}}]
Out[3400]= 7

In[3795]:= SetAttributes1[x_, y_] :=
ToExpression["SetAttributes[SetAttributes, Listable];
SetAttributes[" <> ToString[x] <> ", " <> ToString[y] <> "];
ClearAttributes[SetAttributes, Listable]"]

In[3796] := t = {x, y, z}; SetAttributes1[t, Listable];
Map[Attributes, Flatten[{t, SetAttributes}]]
Out[3796]= {{Listable}, {Listable}, {Listable}, {HoldFirst, Protected}}
```

In the previous fragment a string structure***Headd*** has been presented which represents definition of the**Head** procedure of the same name with standard **Head** function with expansion of functionality of the last. As an example the call**RepStandFunc[Headd, Head,** {{{a},{b,c},{d}}}] is presented whose result is a modification of the**Head*(Headd)*** function whose once-only application to a list of***NestList*–**type returns the heading***NestList*** on such list, whereas the**Head** function on this list returns the heading***List.*** Modifications of the **Head** procedure in string structure***Headd*** are quite simple*(by an appropriate extension of the lists represented by local variables**b**and**c**),* in principle allowing to expand the list of headings arbitrarily widely. However, these headings aren't distinguished by the**Mathematica** as components of***"x_h"*** patterns as the fragment example with function***G*** very visually illustrates. Moreover, this result takes place both at using of the**RepStandFunc** procedure, and at the*prolonged* replacement*(for the duration of the**Mathematica**current session)* of the standard**Head** function by its modification which is located in string structure***Headd.*** As a result of similar procedure the**Mathematica** restart is required for recovery of the original version of the**Head** function if before, it wasn't kept in a datafile of***mx*–**format from which it could be loaded into the current session as that the**RepStandFunc** procedure does. At that it is supposed that a block/procedure/function replacing a standard function***x*** shouldn't contain calls of the initial function***x;*** otherwise, emergence of the special or erroneous situations up to the looping is a quite real, demanding the restart of the**Mathematica** system.
At last, the**SetAttributes1** function completes the previous fragment***;*** its call **SetAttributes1[*x*,*y*]** expands the standard**SetAttributes** function onto the form of presentation of the*first* argument***x*,** for which the*indexed* variables, lists, etc. can be used, for example, providing the ascribing of attributes***y*** to elements of a list***x*.** Meanwhile, the above mechanism of*once-only* use of the substitutes of the same name of standard functions in certain cases is rather effective method, however prolongation of such substitutes on the current session can cause conflict situations with its functions that

significantly use originals of the replaced means. So, the given mechanism should be used a rather circumspectly.

The question of processing of the *formal* arguments with good reason can be considered as the first problem relating to the calculation of tuple of formal arguments of the user functions/modules/blocks that have been activated in the current session directly or on the basis of download of the packages containing their definitions. In the previous works [30-33,48] certain means for the solution of this task have been offered in the form of the procedures **Args, Args0, Args1, Args2,** below we will present similar means in narrower assortment and with the improved functional characteristics. First of all, as a very useful means, we will present the**Args** procedure whose call**Args[*x*]** returns the list of formal arguments of the user module/block/function***x*.** The following fragment represents sourse code of the**Args** procedure with the most typical examples of its usage.

In[2322] **:= V := Compile[{{x, _Real}, {y, _Real}}, (x^3 + y)^2];**
Kr := (#1^2 + #2^4–90*#3) &; H[x_] := Block[{}, x]; Art := Function[{x, y}, x*Sin[y]]; P[y_] := Module[{}, y]; P[x__] := Plus[Sequences[{x}]]; H[x_, y_] := x + y; GS[x_ /; IntegerQ[x], y_ /; IntegerQ[y]] := Sin[90] + Cos[42]; Sv[x_ /; IntegerQ[x], y_ /; IntegerQ[y]] := x^2 + y^2; Sv = Compile[{{x, _Integer}, {y, _Real}}, (x + y)^6]; S := Compile[{{x, _Integer}, {y, _Real}}, (x + y)^3];
G = Compile[{{x, _Integer}, {y, _Real}}, (x + y)];
T := Compile[{{x, _Real}}, (x + y)]; SetAttributes[H, Protected];

In[2323]:= **Args[P_, z___] := Module[{a, b, c, d = {}, k = 1, Vt}, If[CompileFuncQ[P] || BlockFuncModQ[P],**

Vt[y_/; ListQ[y]] := Module[{p = 1, q = {}, t}, While[p <= Length[y], q = Append[q, t = ToString[y[[p]]]; StringTake[t, {1, StringPosition[t, "_"][[1]][[1]]–1}]]; p++]; q]; If[CompileFuncQ[P],

a = StringSplit[ToString[InputForm[Definition2[P]]], "\n \n"][[1]]; b = Quiet[SubStrSymbolParity1[a, "{", "}"]]; b = Select[b, ! StringFreeQ[#, "_"]||! StringFreeQ[a, " Function[" <> #] &]; b = Mapp[StringSplit, b, ", "];

b = Mapp[StringReplace, b, {"{"–> "", "}"–> ""}]; b = Mapp[Select, b, StringFreeQ[#, "Blank$"] &]; c = b[[2]];

For[k, k <= Length[c], k++, d = Append[d, c[[k]] <> b[[1]][[k]]]]; d = ToExpression[d];
If[{z}== {}, d, Flatten[Map[Vt, {d}]]],
If[BlockFuncModQ[P], a = Flatten[{HeadPF[P]}];

For[k, k <= Length[a], k++, d = Append[d,
If[{z}!= {}, Vt[ToExpression["{" <> StringTake[a[[k]], {StringLength[ToString[P]] + 2,–2}] <> "}"]], ToExpression["{" <> StringTake[a[[k]], {StringLength[ToString[P]] + 2,–2}] <> "}"]]]]; If[Length[d] == 1, d[[1]], d],
a = StringTake[StringReplace[ToString[InputForm[Definition2[P]]], "Definition2["–> "", 1], {1,–2}]; If[SuffPref[a, "Function[{", 1],
b = SubStrSymbolParity1[a, "{", "}"]; b = Select[b, ! StringFreeQ[a, "Function[" <> #] &][[1]]; a = StringSplit[StringTake[b, {2,–2}], ", "], a = StringReplace[a, "#"–> "$$$$$"]; a = Map[ToString, UnDefVars[ToExpression[a]]]; Map[ToString,

ToExpression[Mapp[StringReplace, a, “$$$$$”–> “#”]]]]]], $Failed]]

In[2324] **:= Map[Args, {V, S, Sv, T}]**
Out[2324]= {{x_Real,y_Real},{x_Integer,y_Real},{x_Integer,y_Real},{x_Real}}
In[2325]**:= Mapp[Args, {V, S, Sv, T}, gs]**
Out[2325]= {{”x“, “y”}, {”x“, “y”}, {”x“, “y”}, {”x”}}
In[2326]**:= Map[Args, {H, P, GS}]**
Out[2326]= {{{x_},{x_,y_}},{{y_},{x__}},{x_/; IntegerQ[x],y_/; IntegerQ[y]}}
In[2327]**:= Mapp[Args, {H, P, GS}, gs]**
Out[2327]= {{{”x”}, {”x“, “y”}}, {{”y”}, {”x”}}, {”x“, “y”}}
In[2328]**:= Map[Args, {Art, Kr}]**
Out[2328]= {{”x“, “y”}, {“#1“, “#2“, “#3”}}
In[2329]**:= Mapp[Args, {Art, Kr}, gs]**
Out[2329]= {{”x“, “y”}, {“#1“, “#2“, “#3”}}
In[2330]**:= Map[Args, {avz, 50090, a + b}]**
Out[2330]= {$Failed, $Failed, $Failed}

In[2556]**:= Args1[x_ /; BlockFuncModQ[x]] := Module[{b = 1, c = {}, d, p, t, a = Flatten[{PureDefinition[x]}]}, For[b, b <= Length[a], b++, t = ToString[Unique[“agn”]]; p = t <> ToString[x]; ToExpression[t <> a[[b]]]; d = Unique[“avz”];**

AppendTo[c, {Args[p], BlockFuncModQ[p, d]; d}]; d = ToUpperCase[d]; ToExpression[“Clear[” <> p <> “,” <> t <> “,” <> d <> “]”]]; If[Length[c] == 1, c[[1]], c]]

In[2557] **:= Args1[H]**
Out[2557]= {{{x_}, “Block”}, {{x_, y_}, “Function”}}
In[2558]**:= Args1[GS]**
Out[2558]= {{x_ /; IntegerQ[x], y_ /; IntegerQ[y]}, “Function”} In[2559]**:= Args1[P]**
Out[2559]= {{{y_}, “Module”}, {{x__}, “Function”}}

At that, the format of the result returned by a call**Args[*x*]** is defined by type of an object***x*,** namely**:**

– the list of formal arguments with the types ascribed to them is returned on the**Compile** function**;**
– the list of formal arguments with the tests for an admissibility of the actual arguments ascribed to them or without them is returned on {***module, block, typical function***}**;** at that, the**Args** procedure processes the situation***”****objects of the same name with various headings****“*,** returning the nested list of the formal arguments concerning all subobjects composing an object***x*** in the order that is determined by the call**Definition2[*x*];**
– the list of slots {***#1,…,#n***} in string format of formal arguments is returned on pure function in short format while for standard pure function the list of formal arguments in string format is returned**.**

Moreover, the call **Args[*Wg, h*]** with the second optional argument***h –****any admissible expression or any their sequence* **–** returns the result similar to the call with the first argument, with that difference that all formal arguments are encoded in string format, but without types ascribed to arguments and tests for admissibility. On an inadmissible actual

argument the call**Args[*x*]** returns**$Failed.** Told very visually is looked through in the examples which are represented in the previous fragment.
At that, the definition of the**Args** along with standard means uses and our means such as **BlockFuncModQ, CompileFuncQ, Definition2, SuffPref, HeadPF, Mapp, SubStrSymbolParity1** and**UnDefVars** that are considered in the present book and in [28-33]. This procedure is used quite widely, first of all, in problems of system programming in the**Mathematica,** significantly expanding the above–mentioned procedures**Args, Args0, Args1, Args2.** In the same context we will note that a number of the means presented in [32] are absent in the present book because of replacement their by more quick– acting and functionally developed means**;** like this, the**ArgsProc** procedure whose functions are overlapped by the**Args** procedure.

The **Args1** procedure completes the previous fragment, whose call**Args1[*x*]** returns simple or the nested list, whose elements are***2***–element lists, whose *first* element represents the list of formal arguments with the types and tests, ascribed to them while the second**–** an object type in the context {*”Module“*, *“Block”*,*”Function”*}. As argument***x*** the objects on which**BlockFuncModQ[*x*]** returns*True* are allowed. On an unacceptable argument***x*** the procedure call **Args1[*x*]** is returned unevaluated.
The**ArgsBFM** procedure is quite useful means in addition to the procedures **Args** and**Args1;** it is intended for evaluation of formal arguments of a block/ function/module. The next fragment represents source code of the procedure **ArgsBFM** along with typical examples of its usage.

```
In[2396] := ArgsBFM[x_ /; BlockFuncModQ[x], y___] := Module[{b, c= {}, p, a = Flatten[{HeadPF[x]}], d = {}, n = ToString[x] <> “[“, k = 1}, b = Map[ToExpression[“{” <> StringTake[#, {StringLength[n] + 1,–2}] <> “}“] &, a]; c = Map[Map[ToString, #] &, b]; While[k <= Length[c], p = c[[k]]; AppendTo[d, Map[StringTake[#, {1, Flatten[StringPosition[#, “_”]][[1]]–1}] &, p]]; k++]; If[{y}!= {}&& ! HowAct[y], y = c]; d]

In[2397] := G[x_ /; IntegerQ[x], y_ /; IntegerQ[y]] := x + y; G[x_, y__] := x + y; G[x_, y_ /; IntegerQ[y], z_] := x + y + z; G[x_Integer, y__] := x + y; G[x_ /; x == {42, 47, 67}, y_ /; IntegerQ[y]] := Length[x] + y; G[x_ /; IntegerQ[x]] := x

In[2398]:= ArgsBFM[G]
Out[2398]= {{”x“, “y”}, {”x“, “y“, “z”}, {”x“, “y”}, {”x“, “y”}, {”x“, “y”}, {”x”}}

In[2399] := ArgsBFM[G, gs]
Out[2399]= {{”x“, “y”}, {”x“, “y“, “z”}, {”x“, “y”}, {”x“, “y”}, {”x“, “y”}, {”x”}}
In[2400]:= gs
Out[2400]= {{”x_ /; IntegerQ[x]“, “y_ /; IntegerQ[y]”},

{ “x_”, “y_ /; IntegerQ[y]“, “z_”}, {”x_Integer“, “y__”}, {”x_ /; x== {42, 47, 67}“, “y_ /; IntegerQ[y]”}, {”x_”, “y__”}, {”x_ /; IntegerQ[x]”}}
```

The procedure call **ArgsBFM[*x*]** returns the list of*formal arguments* in string format of a block/function/module***x*** whereas the call**ArgsBFM[*x, y*]** with the second optional argument***y*** *–an undefinite variable–* in addition returns thru it the list of formal arguments of the block/function/module***x*** with the tests for admissibility in string format that are ascribed to them.

The next **ArgsTypes** procedure serves for testing of the formal arguments of a block/function/module activated in the current session. The procedure call**ArgsTypes[*x*]** returns the nested list, whose***2*–**element sublists in string format define names of formal arguments and their admissible types*(and in a broader sense the tests for their admissibility along with initial values by default)* respectively. At absence for an argument of type it is defined as***"Arbitrary"*** that is characteristic for arguments of*pure functions* and arguments without the tests and/or initial values ascribed to them, and also which have format patterns {**"__", "___"**}. The following fragment represents source code of the **ArgsTypes** procedure along with typical examples of its usage.

In[2775] **:= V := Compile[{{x, _Real}, {y, _Real}}, (x^3 + y)^2];**
Kr := (#1^2 + #2^4–90*#3) &; H[x_] := Block[{}, x]; Art := Function[{x, y}, x*Sin[y]]; H[x_, y_] := x + y; P[x__] := Plus[Sequences[{x}]]; GS[x__] := x;
P[x_ /; StringQ[x], y_] := StringLength[x];
GS[x_ /; IntegerQ[x], y_ /; IntegerQ[y]] := Sin[90] + Cos[42]; Sv[x_ /; IntegerQ[x], y_ /; IntegerQ[y]] := x^2 + y^2; Sv = Compile[{{x, _Integer}, {y, _Real}}, (x + y)^6]; S := Compile[{{x, _Integer}, {y, _Real}}, (x + y)^3];
G = Compile[{{x, _Integer}, {y, _Real}}, (x + y)];
P[y_] := Module[{}, y]; P[x__] := Plus[Sequences[{x}]]; T := Compile[{{x, _Real}}, (x + y)]; GS[x_, y_String] := {x, y} In[2776]:= **ArgsTypes[x_ /; CompileFuncQ[x] || BlockFuncModQ[x]] :=**

Module[{a = Args[x], c = {}, d = {}, k = 1}, If[CompileFuncQ[x], a = Mapp[StringSplit, Map[ToString, a], "_"]; If[Length[a] == 1, a[[1]], a], If[PureFuncQ[x], a = Map[{#, "Arbitrary"}&, a]; If[Length[a] == 1, a[[1]], a], SetAttributes[ToString, Listable]; a = Map[ToString, a]; ClearAttributes[ToString, Listable]; a = If[NestListQ[a], a, {a}]; For[k, k <= Length[a], k++, c = Append[c,

Mapp[StringSplit, Mapp[StringSplit, a[[k]], "_ /; "], {"___", "__", "_"}]]]; c; For[k = 1, k <= Length[c], k++, d = Append[d, Map[Flatten, c[[k]]]]]; c = {}; For[k = 1, k <= Length[d], k++, c = Append[c, Map[If[Length[#] == 1, {#[[1]], "Arbitrary"}, {#[[1]], StringReplace[#[[2]], "\"–> ""]}] &, d[[k]]]]]; c = Map[If[Length[#] == 1, #[[1]], #] &, c]; If[Length[c] == 1, c[[1]], c]]]]

In[2777] **:= Map[ArgsTypes, {GS, Args}]**
Out[2777]= {{{{"x", "IntegerQ[x]"}, {"y", "IntegerQ[y]"}},
{{"x", "Arbitrary"}, {"y", "String"}}, {"x", "Arbitrary"}}, {{"P", "Arbitrary"}, {"z", "Arbitrary"}}}
In[2778]**:= ArgsTypes[P]**
Out[2778]= {{{"x", "StringQ[x]"}, {"y", "Arbitrary"}},
{"y", "Arbitrary"}, {"x", "Arbitrary"}}
In[2779]**:= Map[ArgsTypes, {Art, Kr}]**
Out[2779]= {{{"x", "Arbitrary"}, {"y", "Arbitrary"}},
{{"#1", "Arbitrary"}, {"#2", "Arbitrary"}, {"#3", "Arbitrary"}}}
In[2780]**:= Map[ArgsTypes, {V, Sv, S, G, T}]**
Out[2780]= {{{"x", "Real"}, {"y", "Real"}}, {{"x", "Integer"}, {"y", "Real"}}, {{"x", "Integer"}, {"y", "Real"}},
{{"x", "Integer"}, {"y", "Real"}}, {"x", "Real"}}

Moreover, the **ArgsTypes** procedure successfully processes the mentioned situation*"objects of the same name with various headings"*, returning the nested *2*–element lists of formal arguments concerning the subobjects composing an object*x*, in the order determined by the**Definition** function. And in this case*2*–element lists have the format, represented above whereas for objects with empty list of formal arguments the empty list is returned, i.e. {}. Unlike **ArgsTypes** of the same name [29,30] the given procedure processes blocks/ modules/functions, including*pure functions* and**Compile** functions. At that, the call**ArgsTypes[*x*]** on an illegal argument*x* is returned unevaluated. Multiple patterns of formats*x*__ and*x*___ allow to determine any number of admissible factual arguments of a block/function/module**;** at that, if the first pattern defines not less than one argument, the second pattern allows absence of the actual arguments. The mentioned patterns formats of formal arguments allow to determine the objects of the specified type, allowing any number of the actual arguments at their calls. This circumstance is the basis for programming of the means that define arity of the user block/function/ module, i.e. number of the actual arguments allowed at the object calls of a specified type that doesn't cause special*(unevaluated calls)* or the erroneous situations caused by discrepancy between number of the received factual arguments and of admissible at determining of an object. The question of calculation of arity of the user block/function/module is rather important in many appendices and, first of all, of system character, but**Mathematica** has no means for its solution therefore certain procedures for the solution of the question have been created such as**Arity, Arity1, Arity2, ArityM, ArityPF** that solve this problem with one or another degree of generality [30-32]. The next fragment presents source code of the**Arity** procedure that generalizes all above–mentioned means solving the arity problem along with examples of its more typical applications.

```
In[2565] := V := Compile[{{x, _Real}, {y, _Real}}, (x^3 + y)^2];
Kr := (#1^2 + #2^4–500*#3) &; H[x_, y_] := x + y; Art := Function[{x, y}, x*Sin[y]];
H[x_] := Block[{}, x]; P[x__] := Plus[Sequences[{x}]]; SetAttributes[H, Protected];
GS[x_ /; IntegerQ[x], y_ /; IntegerQ[y]] := Sin[500] + Cos[42]; Sv[x_ /; IntegerQ[x],
y_ /; IntegerQ[y]] := x^2 + y^2; Sv = Compile[{{x, _Integer}, {y, _Real}}, (x + y)^6];
S := Compile[{{x, _Integer}, {y, _Real}}, (x + y)^3];
G = Compile[{{x, _Integer}, {y, _Real}}, (x + y)];
P[y_] := Module[{}, y]; T := Compile[{{x, _Real}}, (x + y)]; Vs[x_ /; SameQ[{x}, {}]]
:= {x}; W[x_] := x; W[x_, y_] := x + y; W[x_, y_, z_, t_] := Module[{}, x*y*z*t;
W[x_, y_Integer] := x + y

In[2566] := Arity[P_ /; SystemQ[P] || CompileFuncQ[P] || PureFuncQ[P]
||BlockFuncModQ[P]] := Module[{a}, If[SystemQ[P], "System", a = Args[P];
Mapp[SetAttributes, {ToString, StringFreeQ}, Listable]; a = Map[ToString, a]; a =
Map[If[DeleteDuplicates[StringFreeQ[#, "__"]] === {True}, Length[#],
"Undefined"] &, If[NestListQ[a], a, {a}]]; Mapp[ClearAttributes, {ToString,
StringFreeQ}, Listable]; If[Length[a] == 1, a[[1]], a]]]

In[2567] := Map[Arity, {V, S, Sv, T}]
Out[2567]= {2, 2, 2, 1}
In[2568]:= Map[Arity, {H, P, GS}]
Out[2568]= {{1, 2}, {1, "Undefined"}, 2, 1}
In[2569]:= Map[Arity, {Art, Kr, ProcQ, Sin, For}]
```

Out[2569]= {2, 3, 1, "System", "System"}
In[2570]:= **Map[Arity, {avz, 500, a + b}]**
Out[2570]= {Arity[avz], Arity[500], Arity[a+ b]}
In[2571]:= **Arity[W]**
Out[2571]= {4, 2, 1, 2}

In[2666]:= **Arity1[P_ /; SystemQ[P] || CompileFuncQ[P] || PureFuncQ[P] || BlockFuncModQ[P]] :=**

Module[{a}, If[SystemQ[P], "System", a = Args[P]; a = Mapp1[ToString, a]; a = Map[If[DeleteDuplicates[StringFreeQ[#, "__"]] === {True}, Length[#], "Undefined"] &, If[NestListQ[a], a, {a}]]; If[Length[a] == 1, a[[1]], a]]] In[2667]:= **Map[Arity1, {V, S, Sv, T}]**

Out[2667] = {2, 2, 2, 1}
In[2668]:= **Map[Arity1, {H, P, GS}]**
Out[2668]= {{1, 2}, {1, "Undefined"}, 2, 1}
In[2669]:= **Map[Arity1, {Art, Kr, ProcQ, Sin, For}]**
Out[2669]= {2, 3, 1, "System", "System"}
In[2670]:= **Arity1[W]**

Out[2670] = {4, 2, 1, 2}
In[2671]:= **Map[Arity1, {avz, 500, a + b}]**
Out[2671]= {Arity1[avz], Arity1[500], Arity1[a+ b]}

On blocks /functions/modules with undefinite number of arguments the call **Arity[*x*]** returns**"***Undefined***",** on the system functions the call**Arity[*x*]** returns ***"System"*** while on the objects having the fixed number of actual arguments their number is returned, in other cases the call is returned unevaluated. We will note that the**Arity** procedure processes the special situation**"***objects of the same name with various headings***",** returning the list of arities of subobjects composing an object***x*.** At that, between this list and the list of definitions of subobjects which is returned on the call**Definition[*x*]** there is one–to–one correspondence. The definition of the**Arity** procedure along with standard means uses and our means such as**Args, BlockFuncModQ, CompileFuncQ, Mapp, SystemQ, PureFuncQ, NestListQ** that are considered in the present book and in [28-33]. Moreover, at programming of the**Arity** in the light of simplification of its algorithm is expedient for the period of a procedure call to ascribe to the system functions**ToString** and**StringFreeQ** the attribute ***Listable*,** allowing to considerably reduce source code of the**Arity.** At last, the**Arity1** procedure**–***a rather effective equivalent analog of***Arity***procedure***–** completes the previous fragment.

The **ArityBFM** procedure defining arity of objects of type {***module, classical function, block***} serves as quite useful addition to the procedures**Arity** and **Arity1.** The following fragment represents source code of the**ArityBFM** and the most typical examples of its usage.

In[2474]:= **ArityBFM[x_ /; BlockFuncModQ[x]] := Module[{b, a = Flatten[{HeadPF[x]}]}, b = Map[If[! StringFreeQ[#, {"__", "___"}], "Undefined",**

Length[ToExpression[" {" <> StringTake[StringReplace[#, ToString[x] <> "["–> "", 1], {1,–2}] <> "}"]]] &, a]; If[Length[b] == 1, b[[1]], b]] In[2475]:= **G[x_ /;**

IntegerQ[x], y_ /; IntegerQ[y]] := x + y;

G[x_Integer, y__] := x*y; G[x_, y_ /; IntegerQ[y], z_] := x + y + z; G[x_, y__] := x + y; G[x_ /; IntegerQ[x]] := x;
G[x_ /; x == {42, 47, 67}, y_ /; IntegerQ[y]] := Length[x] + y; In[2476]:= **ArityBFM[G]**
Out[2476]= {2, 3, "Undefined", 2, "Undefined", 1}
In[2477]:= **S[x_, y_] := x*y; V[x__] := {x}; Map[ArityBFM, {S, V}]** Out[2477]= {2, "Undefined"}
In[2478]:= **V[a_, b_, c_, d_, h_] := N[h*(3*a*b + (c–b)*d + (d–a)*c)/3000]** In[2479]:= **{V[25, 18, 47, 72, 67], ArityBFM[V]}**
Out[2479]= {126.116, {5, "Undefined"}}

The procedure call **ArityBFM[*x*]** returns***arity(number of arguments)*** of an object***x*** of type {*block, function, module*}; at that, a function of classical type is understood as function***x*,** i.e. some function with*heading***.** In the presence in heading of formal arguments with patterns {**"__","___"**} arity is defined as undefinite*("Undefined")* because arity is understood as a real number of the factual arguments, admissible at a call of an object***x*** which in that case can be undefinite. At that, on objects***x*** of type, different from the specified, the procedure call is returned unevaluated.

Like the **Maple** system, the**Mathematica** system doesn't give a possibility to test inadmissibility of all actual arguments in a block/function/module in a point of its call, interrupting its call already on the*first* inadmissible actual argument. Meanwhile, in view of importance of definition of all*inadmissible* actual arguments only for one pass, the**TestArgsTypes** procedure solving this important enough problem and presented in our book [30] and package ***AVZ_Package*** [48] has been created. At that, the emergence of new means and updating of our existing functional means allows to update also and the given means, presented by the procedure of the same name. The following fragment presents source code of the**TestArgsTypes** procedure along with one its useful modification**TestArgsTypes1** and examples of their usage.

In[2760]:= **TestArgsTypes[P_ /; ModuleQ[P] || BlockQ[P] ||**

QFunction[P], y_] := Module[{c, d = {}, h, k = 1, a = Map[ToString, Args[P]], b = ToString[InputForm[y]]}, ClearAll["$TestArgsTypes"]; If[! SuffPref[b, ToString[P] <> "[", 1], Return[y], c = Map[ToString1, ToExpression["{" <> StringTake[b, {StringLength[ToString[P]] + 2,–2}] <> "}"]]]; If[Length[a] != Length[c], $TestArgsTypes = "Quantities of formal and factual arguments are different"; $Failed, For[k, k <= Length[a], k++, d = Append[d, ToExpression["{" <> c[[k]] <> "}" <> " /. " <> a[[k]] <> "–> True"]]]; d = Map[If[ListQ[#], #[[1]], #] &, d]; h = Flatten[Map3[Position, d, Cases[d, Except[True]]]]; h = Map[{#, If[ListQ[d[[#]]], Flatten[d[[#]], 1], d[[#]]]}&, h]; $TestArgsTypes = If[Length[h] == 1, h[[1]], h]; $Failed]]

In[2761] **:= P[x_, y_String, z_ /; If[z === 90, True, False]] := {x, y, z}** In[2762]:= **TestArgsTypes[P, P[agn, "ArtKr", 90]]**
Out[2762]= {agn, "ArtKr", 90}
In[2763]:= **TestArgsTypes[P, P[x, y, z]]**

```
Out[2763]= $Failed
In[2764]:= $TestArgsTypes
Out[2764]= {{2, y}, {3, z}}
In[2765]:= TestArgsTypes[P, P[x, y, z, h]]
Out[2765]= $Failed
In[2766]:= $TestArgsTypes
Out[2766]= “Quantities of formal and factual arguments are different” In[2767]:=
TestArgsTypes[P, P[x, “y”, {500}]]
Out[2767]= $Failed
In[2768]:= $TestArgsTypes
Out[2768]= {3, {500}}
In[2769]:= TestArgsTypes[P, P[x, a + b, {500}]]
Out[2769]= $Failed
In[2770]:= $TestArgsTypes
Out[2770]= {{2, a+ b}, {3, {500}}}

In[2771]:= VS[x_, n_ /; IntegerQ[n], y_, z_/; StringQ[z], L_ /; ListQ[L] &&
MemberQ[{{0}, {1}, {0, 1}}, Sort[DeleteDuplicates[Flatten[L]]]]] :=

Block[ {}, L[[StringLength[y <> z] + n]]] In[2772]:= VS[6,–4, “A”, “vz”, {0, {1, 0, 1},
{1, 0, 0, 0, 1, 1, 1, 0, 0, 1}}]

Out[2772] = {1, 0, 0, 0, 1, 1, 1, 0, 0, 1}
In[2773]:= VS[6, 7.2, A, “vz”, {0, {1, 0, 1}, {1, 0, 0, 0, 1, 1, 1, 0, 0, 1}}] Out[2773]=
VS[6, 7.2, A,”vz“, {0, {1, 0, 1}, {1, 0, 0, 0, 1, 1, 1, 0, 0, 1}}] In[2774]:=
TestArgsTypes[VS, VS[9, 7.2, A, “v”, {0, {1, 0, 1},{1, 0, 1, 1, 0, 1}}]]

Out[2774] = $Failed
In[2775]:= $TestArgsTypes
Out[2775]= {2, 7.2}
In[2776]:= TestArgsTypes[VS, VS[9, 7.2, A, vz, {0, {1, 0, 1}, {2, 0, 0, 0, 7, 2}}]]
Out[2776]= $Failed
In[2777]:= $TestArgsTypes
Out[2777]= {{2, 7.2}, {4, vz}, {5, {0, True, 2, 0, 0, 0, 7, 2}}}
In[2778]:= TestArgsTypes[VS, VS[9, 0, “A”, “v”, {0, {1, 0, 0,1}, {1, 0, 1, 0, 1}}]]
Out[2778]= {1, 0, 0, 1}
In[2779]:= $TestArgsTypes
Out[2779]= $TestArgsTypes

In[2862]:= TestArgsTypes1[P_ /; ModuleQ[P] || BlockQ[P] ||

QFunction[P], y_] := Module[ {c, d = {}, h, k = 1, n, p, w, w1, a =
Quiet[ArgsTypes[P]], g = Map[ToString1, Args[P]], b = ToString[InputForm[y]]}, a
= Map[{#[[1]], StringReplace[#[[2]], “\\”–> ””]}&, a]; ClearAll[“$TestArgsTypes”,
“$$Art$Kr$$”]; If[! SuffPref[b, ToString[P] <> “[“, 1], Return[y], c =
Map[ToString1, ToExpression[“{” <> StringTake[b, {StringLength[ToString[P]] +
2,–2}] <> “}“]]]; If[Length[a] != Length[c], Return[$TestArgsTypes = “Quantities of
formal and factual arguments are different”; $Failed], w = Map[StringTake[#, {1,
StringPosition[#, “_”][[1]][[1]]–1}] &, g]; w1 = Map[ToString, Unique[w]]; While[k
```

<=Length[w], ToExpression[w1[[k]] <> " = " <> w[[k]]]; k++]; Map[ClearAll, w]; For[k = 1, k <= Length[a], k++, p = a[[k]]; If[p[[2]] === "Arbitrary", d = Append[d, True],

If[StringFreeQ[g[[k]], " /; "],

If[ToExpression["Head[" <> c[[k]] <> "] === " <> p[[2]]], d = Append[d, True], d = Append[d, False]], $$Art$Kr$$ = ToExpression[p[[1]]]; n = ToExpression[{p[[1]] <> " = " <> c[[k]], p[[2]]}]; ToExpression[p[[1]] <> " = " <> "$$Art$Kr$$"]; If[n[[–1]], d = Append[d, True], d = Append[d, False]]]]]]; h = DeleteDuplicates[Flatten[Map3[Position, d, Cases[d, Except[True]]]]]; h = Map[{#, If[ListQ[c[[#]]], Flatten[c[[#]], 1], c[[#]]]}&, h]; $TestArgsTypes = If[Length[h] == 1, h[[1]], h]; k = 1; While[k <= Length[w], ToExpression[w[[k]] <> " = " <> w1[[k]]]; k++]; ClearAll["$$Art$Kr$$"]; $Failed]

In[2863] **:= TestArgsTypes1[P, P[x, a + b, {500}]]**
Out[2863]= $Failed
In[2864]**:= $TestArgsTypes**
Out[2864]= {{2, "a+ b"}, {3, "{500}"}}
In[2865]**:= TestArgsTypes1[P, P[agn, "ArtKr", 90]]**
Out[2865]= {agn, "ArtKr", 90}
In[2866]**:= TestArgsTypes1[P, P[x, y, z, h]]**
Out[2866]= $Failed
In[2867]**:= $TestArgsTypes**
Out[2867]= "Quantities of formal and factual arguments are different" In[2868]**:= TestArgsTypes1[P, P[x, y, z]]**
Out[2868]= $Failed
In[2869]**:= $TestArgsTypes**
Out[2869]= {{2, "y"}, {3, "z"}}
In[2870]**:= TestArgsTypes1[VS, VS[9, 7.2, A, vz, {0, {1, 0, 1}, {2, 0, 1, 5, 6, 2}}]]**
Out[2870]= $Failed
In[2871]**:= $TestArgsTypes**
Out[2871]= {{2, "7.2"}, {4, "vz"}, {5, "{0, {1, 0, 1}, {2, 0, 1, 5, 6, 2}}"}}

In[2920] **:= TestArgsTypes2[x_ /; ModuleQ[x]||BlockQ[x]||QFunction[x], y__] := Module[{a = Quiet[ArgsTypes[x]], b = Map[ToString1, {y}], c = {y}, d = {}, k = 1, p}, If[Length[c] != Length[a],**

"Quantities of formal and factual arguments are different", For[k, k <= Length[c], k++, p = a[[k]];
AppendTo[d, If[p[[2]] === "Arbitrary", True, If[SymbolQ[p[[2]]], ToString[Head[c[[k]]]] === p[[2]],

ToExpression[StringReplace[p[[2]], {"[" <> p[[1]] <> "]"–> "[" <> b[[k]] <> "]", " " <> p[[1]] <> " "–> " " <> b[[k]] <> " ", " " <> p[[1]] <> "]" -> " " <> b[[k]] <> "]"}]]]]]];

If[MemberQ[d, False], Partition[Riffle[{y}, d], 2], {True, x[y]}]]]

In[2921] **:= TestArgsTypes2[VS, 90, 50]**
Out[2921]= "Quantities of formal and factual arguments are different" In[2922]**:= F[x_,**

y_String, z_Integer, t_ /; ListQ[t]] :=

Module[{}**, x*z + StringLength[y]*Length[t]]** In[2923]**:= TestArgsTypes2[F, 90, 500, 72, a + b]**
Out[2923]= {{90, True}, {500, False}, {72, True}, {a+ b, False}}
In[2924]**:= TestArgsTypes2[F, 50, "Agn", 500,** {**r, a, n, s**}**]**
Out[2924]= {True, 25012}
In[2925]**:= TestArgsTypes2[P, x, y, z]**
Out[2925]= {{x, True}, {y, False}, {z, False}}

In[2932]**:= TrueCallQ[x_ /; BlockFuncModQ[x], y__] :=**
Quiet[Check[If[UnevaluatedQ[x, y], False, x[y]; True], False]]

In[2933] **:= TrueCallQ[VS, 9, 7.2, A, vz,** {**0,** {**1, 0, 1**}**,** {**2, 0, 1, 5, 6, 2**}}**]** Out[2933]= False
In[2934]**:= TrueCallQ[P, x, y, z, h]**
Out[2934]= False
In[2935]**:= TrueCallQ[VS, 9, 7.2, A, "vz",** {**0,** {**1, 0, 1**}**,** {**1, 0, 0, 0, 1, 1, 1, 0, 0, 1**}}**]**
Out[2935]= False
In[2936]**:= TrueCallQ[P, agn, "ArtKr", 90]**
Out[2936]= True

Call of the above procedure **TestArgsTypes[*x*, *x*[…]]** processes a procedure ***x*** call in way that returns result of a procedure call***x*[…**] in case of absence of inadmissible actual arguments and equal number of the factual and formal arguments in a point of procedure call***x;*** otherwise**$Failed** is returned. At that through the global variable**$TestArgsTypes** the nested list is returned, whose two-element sublists define the set of inadmissible actual arguments, namely**:** the*first* element of a sublist defines number of inadmissible actual argument while the*second* element**–** its value. At discrepancy of number of formal arguments to number of actual arguments through**$TestArgsTypes** the appropriate diagnostic message is returned**,** namely**:*"Quantities of formal and factual arguments are different"*.**

Meanwhile, for simplification of the testing algorithm realized by the above procedure it is supposed that formal arguments of a certain procedure***x*** are typified by the pattern**"_"** or by construction***"Argument_/;Test".*** Moreover, it is supposed that the*unevaluated* procedure call***x*** is caused by discrepancy of types of the actual arguments to the formal arguments or by discrepancy of their quantities only. So, the question of testing of the actual arguments is considered at the level of the heading of a block/function/module only for a case when their number is fixed. If a procedure/function allows optional arguments, their typifying assumes correct usage of any expressions as the actual values, i.e. the type of the format***"x_"*** is supposed. In this regard at necessity, their testing should be made in the body of a procedure/function as it is illustrated by useful enough examples in [32]. So, at difficult enough algorithms of check of the received actual arguments onto admissibility it is recommended to program them in the*body* of blocks/modules what is more appropriate as a whole.

Meanwhile, as an expansion of the**TestArgsTypes** procedure the possibility of testing of the*actual* arguments onto*admissibility* on condition of existence in headings of formal arguments of types {***"x__", "x___"***} can be considered. The receptions, used in the**TestArgsTypes1** procedure which is one useful modification of the above**TestArgsTypes** procedure is a rather perspective prerequisite for further expansion of functionality of these means. A result of call**TestArgsTypes1[*x*,*x*[…]]** is similar to the call**TestArgsTypes[*x*,*x*[…]]** only with difference that values of inadmissible actual arguments are given in string format. At that without reference to smaller reactivity of the*second* procedure, the algorithm used at its programming is rather interesting for a number of applications, first of all, of system character**;** its analysis can be a rather useful to the interested reader who wishes to learn***Math*–**language more deeply. This remark concerns some other means of the present book.

Meanwhile, it must be kept in mind that use by procedures **TestArgsTypes** and**TestArgsTypes1** of the global variable**$TestArgsTypes** through which information on the inadmissible actual arguments received by a tested*block/ procedure* at its calls is returned, should be defined in the user**'**s package that contains definitions of these procedures, i.e. to be predetermined, otherwise diagnostic information isn**'**t returned thru it. It can be done, for example, by means of inclusion in the***AVZ_Package*** package of the following block**:**

```
Begin["`$TestArgsTypes`"]
$TestArgsTypes = 50090
End[]
```

with obligatory providing a reference*(usage)* for this variable, for example, of the kind**:** **$TestArgsTypes::usage = "**The global variable **$TestArgsType** defined by the procedures**TestArgsTypes** and**TestArgsTypes1."**

This remark should be considered at programming of the procedures which use the global**$–**variables for additional return of results, i.e. such variables should be initiated, in particular, in a package containing the definitions of means with their usages.

In some cases the **TestArgsTypes2** procedure which is a modification of the previous procedures**TestArgsTypes** and**TestArgsTypes1** is a rather useful means**;** the call**TestArgsTypes2[*P*, *y*],** where***P* –** a block, function with the heading, or module, and***y* –** a nonempty sequence of the actual arguments passed to the***P*,** returns the list of the format {***True*,*P*[*y*]**} if all arguments***y*** are admissible**;** the call returns the nested list whose elements are ***the call returns the nested list whose elements are***element sublists whose first element defines an actual argument whereas the second element defines its admissibility {***True*,*False***}**;** at last, in case of discrepancy of quantities of formal and actual arguments the next message is returned**:** ***"****Quantities of formal and factual arguments are different****"*.** The above fragment contains source code of the**TestArgsTypes2** procedure with examples.

At last, in contrast to the above procedures**TestArgsTypes TestArgsTypes2** that provide the differentiated testing of the actual arguments received by a tested object for their admissibility, the simple function**TrueCallQ** provides testing of correctness of the call of an object of type {*Block***,***Function***,***Module*} as a whole**;** the call**TrueCallQ[*x*, *arg*]** returns*True* if the call***x*[*arg*]** is correct, and*False* otherwise. At that, the lack of the fact of the unevaluated call, and lack of the special or erroneous situations distinguished by**Mathematica** is understood as a correctness of the call. The source code of the function with typical examples of its use completes the previous fragment. It is necessary to note the interesting possibilities of further development of the procedures **TestArgsTypes–TestArgsTypes2** in a number of the important directions, in particular, in case of variable number of the actual arguments, which we leave to the interested reader. To the procedures **TestArgsTypes – TestArgsTypes2** and the **TrueCallQ** function in a certain degree the**TestArgsCall** procedure adjoins whose call allows to allocate definitions of a block/function or a module on which the call with the given actual arguments is quite correct. The following fragment represents source code of the**TestArgsCall** procedure with typical examples of its application.

In[2889]:= **TestArgsCall[x_ /; BlockFuncModQ[x], y___] := Module[{d, p, h = {}, k = 1, a = Flatten[{PureDefinition[x]}], b = Flatten[{HeadPF[x]}], c = "$$$", n = ToString[x]},**

While[k <= Length[b], d = c <> n; ToExpression[c <> b[[k]] <> ":=90"]; p = Symbol[d][y]; ToExpression["Clear[" <> d <> "]"]; If[p === 90, AppendTo[h, a[[k]]]]; k++]; If[Length[h] == 1, h[[1]], h]]

In[2890] := **G[x_ /; IntegerQ[x], y_ /; IntegerQ[y]] := x + y;**
G[x_Integer, y__] := x + y; G[x_, y__] := x+y;
G[x_, y_ /; IntegerQ[y], z_] := x+y+z; G[x_ /; IntegerQ[x]] := x; G[x_ /; x == {42, 47, 67}, y_ /; IntegerQ[y]] := Length[x] + y;

In[2891] := **TestArgsCall[G, 19.42, 90]**
Out[2891]= "G[x_, y__]:= x+ y"
In[2892]:= **V[x_ /; IntegerQ[x], y_ /; IntegerQ[y]] := x + y;**

TestArgsCall[V, 19.42, 90] Out[2892]= {}
In[2893]:= **TestArgsCall[Avz, 19.42, 90]**
Out[2893]= TestArgsCall[Avz, 19.42, 90]

The procedure call **TestArgsCall[*x*, *y*]** returns a definition or the definitions list of a block/function/module***x*** on which the call with the tuple of actual arguments***y*** is correct, i.e. their types correspond to admissible types of the formal arguments. Otherwise, the procedure call returns the empty list, i.e. {}; on inadmissible argument***x*,** different from a block/function module, the call is returned unevaluated.

Whereas the procedure call **TestFactArgs[*x*, *y*]** returns the list from*True* and *False* that defines who of the actual arguments determined by a sequence***y*** will be admissible in the call***x*[*y*],** where***x*** **–** an object name with a heading *(block,function,module)***.** The procedure assumes equal number of the formal and actual arguments defined by a sequence***y*,** along with existence for an object***x*** of the fixed number of arguments**;** otherwise, the call**TestFactArgs** returns**$Failed.** The next fragment represents source code of the procedure **TestFactArgs** along with typical examples of its usage.

In[2535]:= **TestFactArgs[x_ /; ProcQ[x] || QFunction[x], y__] :=**

Module[{b, c = {}, d, p = {y}, k = 1, a = Flatten[{HeadPF[x]}][[1]]}, b = StrToList["{" <> StringTake[a, {StringLength[ToString[x]] + 2,–2}] <> "}"]; b = Map[StringSplit[#, "_"] &, b]; If[Length[b] == Length[p] && StringFreeQ[a, "__"],

```
While[k <= Length[b], d = b[[k]]; If[Length[d] == 1, AppendTo[c, True],
If[Length[d] == 2 && SymbolQ[d[[2]]], AppendTo[c, Head[p[[k]]] ===
Symbol[d[[2]]]],
If[SuffPref[d[[2]], " /; ", 1],
AppendTo[c, ToExpression[StringReplace3[ StringTake[d[[2]], {5,–1}], d[[1]],
ToString[p[[k]]]]]]]]]; k++]; c, $Failed]]

In[2536] := VGS[x_, y_Integer, z_ /; ListQ[z]] := Flatten[{x, y, z}] In[2537]:=
TestFactArgs[VGS, avz, 72, {g, s, a, k}]
Out[2537]= {True, True, True}
In[2538]:= TestFactArgs[VGS, 42, ag, a + b]
Out[2538]= {True, False, False}
In[2539]:= TestFactArgs[VGS, 42, ag, a + b, 500]
Out[2539]= $Failed
```

The **TestFactArgs** procedure is a generalization of the***CheckArgs*** procedure of the**Maple** system in case of determination of admissibility of the factual arguments for the blocks/functions/modules. The presented tools of testing of the factual arguments at calls of procedures can be useful enough at the organization of robust program systems of a rather large size. In particular, these means allow rather effectively beforehand to test the correctness of the procedures calls on those or other tuples of factual arguments.

6.6. Local variables of modules and blocks; the means of manipulation by them in the*Mathematica*software

Having considered in the previous two sections the means of manipulation with definitions of the blocks/functions/modules along with their headings, we move on to consideration of means whose circle of interests includes the problems linked with manipulation with the following major component of definitions of blocks and modules– the***local variables.*** So, this component defines the first leading variable in a block/module definition, as a function from two variables– the list of local variables, and its body. Local variables take place only for procedural objects***(Module****and****Block)*** of***Math***language whereas for functions such concept is absent. The*local variables* have only a module body as an area of their action, without crossing with the variables of the same name outside of its environment. Meanwhile, namely between objects of types of {***Block****and****Module***} there is very essential distinction that is based on mechanisms of local variables which are used by both types of objects and which are considered in [30-33] enough in detail. In view of the importance of the given component for which the**Mathematica** has no tools of manipulation it is very desirable to have similar means. Meanwhile, pre has the meaning to consider the given component of modules and blocks in more detail.
First of all, as for admissibility of the*local* variables for traditional functions. The statement about their inadmissibility isn't absolutely right, namely. The local variables shouldn't have crossing with the variables of the same name, outside of the body of an object in which they are defined. Using the given postulate, on the basis of an artificial reception it is possible to solve also this problem. Basic principle of this reception is presented on a simple example.

In[2617]:= **PrevNextVar[x_ /; SymbolQ[x], t_ /; IntegerQ[t], y___] := Module[{a = ToString[x], b, c = {}, d, k, n}, b = Characters[a]; n = Length[b];**

For[k = n, k >= 1, k —, If[IntegerQ[ToExpression[b[[k]]]], AppendTo[c, b[[k]]], d = StringJoin[b[[1 ;; k]]]; Break[]]]; k = ToExpression[c = StringJoin[Reverse[c]]]; If[SameQ[k, Null] || {y}== {}&& k–t <= 0, x, If[c == "", x, ToExpression[d <> ToString[k + If[{y}!= {}, t,–t]]]]]]] In[2618]:= **PrevNextVar[avz1942, 2, 5]**
Out[2618]= avz1944

In[2619] := **PrevNextVar[avz1942, 2]**
Out[2619]= avz1940
In[2620]:= **PrevNextVar[ab90xyz, 5]**
Out[2620]= ab90xyz
In[2621]:= **G[x_, y_] := {ListAssignP[{Unique["a"]}, 1, 72],**

ListAssignP[{Unique["b"]}, 1, 67], PrevNextVar[Unique["a"], 2]*x + PrevNextVar[Unique["b"], 2]*y}[[–1]] In[2622]:= **G[90, 500]**
Out[2622]= 39 980

Above all, we will need the **PrevNextVar** procedure for a special processing of the symbols of the format**<*symbol*><*integer*>** which end with an integer. The procedure call**PrevNextVar[*x, t*]** on a symbol***x*** of the mentioned format **<*symbol*><*integer*>** returns the symbol of the format**<*symbol*><*integer – t*>** while the call**PrevNextVar[*x,t, h*]** where***h* –** an arbitrary expression returns symbol of the format**<*symbol*><*integer+ t*>.** At condition**`*integer – t <= 0*`** or in case of the format***x*** different from the mentioned the source symbol***x*** is returned. The previous fragment represents source code of the procedure with examples of its usage. The given procedure represents as independent interest, and is essentially used for solution of the problem of local variables in case of definition of the user traditional functions.

For solution of the problem of use of *local* variables for functions along with the previous**PrevNextVar** procedure our**ListAssignP** procedure providing assignment of a value to a list element with the given number, and system **Unique** function generating the symbols new for the current session every time at its call are used. The general format of definition of a function with local variables can be presented as follows.

F[*x_, y_,* …] := {ListAssignP[{Unique[*"a1"*]},*1, b1*],
ListAssignP[{Unique[*"a2"*]},*1, b2*],
=========================
ListAssignP[{Unique[*"at"*]},*1, bt*],

BODY[*x, y,* …,PrevNextVar[Unique[*"a1"*],*j*], …, PrevNextVar[Unique[*"at"*],*j*]}[[–*1*]]; *j = kt (k=1..n)*

According to the above format, a function is defined in the form of the list, whose first***t*** elements define the local variables with initial values ascribed to them whereas the body of this function is a certain function from formal arguments and local variables whose each encoding has the following view **PrevNextVar[Unique[*"ap"*],*j*],*j = kt(k=1..n),*** where***n* –** number of usages of a local variable***"ap"*** in the function body. At that, the last element of such list determines result of the call of a function given in similar format as very visually illustrates example of the***G*** function of the previous fragment. So, ehe described artificial reception allows to use local variables in functions, however their opportunities are rather limited. At that, each call of function of this kind generates the mass of variables which aren't crossed with the previous variables of the current session, but they can enough significantly litter the area of variables of the current session**;** in this the reader can rather simply make sure by the means of call**Names["'*"].** Therefore, the artificial possibility of use of local variables by functions, and the expedience of this is not entirely the same. Meanwhile, the presented reception can be a rather useful at programming of certain problems, first of all, of system character. Blocks and modules in the**Mathematica** function as follows. At each call of a module for its local variables the new symbols determining their names, unique in the current session are generated. Each local variable of a module is identified by a symbol of the form***Name$num*** where***Name –*** the name of a*local* variable determined in a module, and***num –*** its current number in the current session. At that, the number is defined by variable***$ModuleNumber*** as it illustrates the following rather simple fragment, namely**:**

```
In[2572] := G[x_, y_, z_] := Module[{a, b, c}, h = a*x + b*y + c*z; {h, a, b,
Symbol["a" <> "$" <> ToString[$ModuleNumber–1]]}]
In[2573]:= {$ModuleNumber, G[72, 67, 47], $ModuleNumber}
Out[2573]= {4477, {72 a$4477+ 67 b$4477+ 47 c$4477, a$4477, b$4477, a$4477},
4478}
In[2574]:= {$ModuleNumber, G[72, 67, 47], $ModuleNumber}
Out[2574]= {4479, {72 a$4479+ 67 b$4479+ 47 c$4479, a$4479, b$4479, a$4479},
4480}
In[2575]:= G[x_, y_, z_] := Block[{a, b, c}, h = a*x + b*y + c*z; {h, a, b, Symbol["a"
<> "$" <> ToString[$ModuleNumber–1]]}]
In[2576]:= {$ModuleNumber, G[72, 67, 47], $ModuleNumber}

Out[2576] = {2386, {72 a+ 67 b+ 47 c, a, b, a$4480}, 4480}
In[2577]:= n = 1; While[n <= 3, Print[$ModuleNumber]; n++] 4489
4489
4489
In[2578]:= {$ModuleNumber, $ModuleNumber}
Out[2578]= {4490, 4490}
```

Of the given example the principle of assignment of current numbers to the local variables quite accurately is traced at each new reference to a module containing them. Also from the fragment follows that increase of the current numbers for local variable blocks at their calls isn't done in view of different mechanisms of processing of modules and blocks. At that, on condition of knowledge of the current numbering for local variables of a module there is an opportunity to dynamically receive their values outside of the module after each its call as illustrates the following rather simple and very evident fragment, namely**:**

```
In[2555]:= S[x_, y_] := Module[{a = $ModuleNumber–1,
b = $ModuleNumber–1, c = $ModuleNumber–1}, h := a*x + b*y + c;

{ h, Symbol["a$" <> ToString[$ModuleNumber–1]], Symbol["b$" <>
ToString[$ModuleNumber–1]], Symbol["c$" <> ToString[$ModuleNumber–1]], a b,
c}] In[2556]:= S[77, 67]
```

Out[2556] = {347420, 2396, 2396, 2396, 5740816, 2396}
In[2557]:= **g := {a$2397, b$2397, c$2397}**
In[2558]:= **S[72, 67]**
Out[2558]= {338520, 2418, 2418, 2418, 5846724, 2418}
In[2559]:= **d := {g, {a$2419, b$2419, c$2419}}**
In[2560]:= **S[72, 67]**
Out[2561]= {339500, 2425, 2425, 2425, 5880625, 2425}
In[2562]:= **{d, {a$2426, b$2426, c$2426}}**
Out[2562]= {{{2396, 2396, 2396}, {2418, 2418, 2418}}, {2425, 2425, 2425}}

Thus, the user has opportunity to work with the local variables and outside of a module containing them, i.e. as a matter of fact at the level of the global variables what in certain cases can be used quite effectively at programming of the different problems and, first of all, of problems of system character.

In[2587]:= **Kr[x_, y_] := Module[{a, b}, h := a*x + b*y; {{a, b, h}, h}]**

In[2588] := **Kr[18, 25]**
Out[2588]= {{a$2436, b$2436, 18 a$2436+ 25 b$2436}, 18 a$2436+ 25 b$2436}
In[2589]:= **%[[1]][[1]]^2 + Take[%[[1]], {2, 2}]^2**
Out[2589]= {a$2436^2+ b$2436^2}

In[2590]:= **Kr[x_, y_] := Module[{a, b}, a = 96; b = 89; h := a*x + b*y; Print[{"a$" <> ToString[$ModuleNumber–1], "b$" <> ToString[$ModuleNumber–1]}]; {Symbol["a$" <> ToString[$ModuleNumber–1]], Symbol["b$" <> ToString[$ModuleNumber–1]]}]** In[2591]:= **Kr[18, 25]**

{a $2446, b$2446}
Out[2591]= {96, 89}
In[2592]:= **%[[1]]^2 + %[[–1]]^2**
Out[2592]= 17 137

The previous simple fragment rather visually illustrates the aforesaid. As a rule, the user shouldn't operate with values of local variables outside of the module; meanwhile, in case of work with a module in the dialogue mode or at using for monitoring of a module performance of a function, for example, **Trace** these local variables are visualized. Moreover, such opportunity can be used for non–standard calculations, only the effect from it is completely defined by experience and skills of the user, his knowledge of the system. In the**Maple** system the similar explicit mechanism of operating with the local variables outside of procedures is absent though the similar mechanism can be realized by some special receptions, in particular, on the basis of so-called method of***"disk transits"*** [22]. However, such approach does variables of a procedure as***really local*** variables with the scope limited by the procedure. In this case local variables are inaccessible outside of the procedure, what in certain respects it is possible to consider as some prerequisite for definition of a*"black box"* and quite natural transition to the paradigm of the modular organization in programming.

In some cases it is necessary to generate object names that are unique to the current session. For this purpose, the afore–mentioned**Unique** function is designed, that generates the names without the attributes ascribed to them. At that, for ensuring of uniqueness of the generated symbols each call of the **Unique** function provides an increment for a value of the system variable ***$ModuleNumber.*** The mechanism of functioning of the**Unique** is similar to the mechanism of generating of names for local variables of a module. The simple example illustrates one of approaches to software realization of the **Unique** by means of the**Un** procedure, whose source code with examples are given below, while useful procedure**Unique2** completes the fragment; the call**Unique2[*x,y*]** returns an unique name in string format that depends on the*second* argument or its absence, at the same time ascribing to the name an arbitrary value***x.***

In[2555]:= **Un[x___] := Module[{a},**

If[{x}== {}, Symbol["$" <> ToString[$ModuleNumber]], a[y_] := If[StringQ[y], Symbol[y <> ToString[$ModuleNumber]], If[Head[y] == Symbol, Symbol[ToString[y] <> "$" <> ToString[$ModuleNumber]], y]]; If[ListQ[x], Map[a, Flatten[x]], a[x]]]]

In[2556]:= {**Un[], Un[S], Un["G"], Un[{x, y, z}], Un[V]**}
Out[2556]= {$1063, S$1064, G1065, {x$1066, y$1066, z$1066}, V$1067}

In[2570] **:= Unique1[x_, y___] := Module[{a = Unique[y], b}, b = ToString[a]; ToExpression[ToString[a] <> "=" <> ToString1[x]]; b]**
In[2571]:= {**Unique1[90, agn], Unique1[500]**}
Out[2571]= {"agn$1086", "$27"}
In[2572]:= **ToExpression[{"agn$1086", "$27"}]**
Out[2572]= {90, 500}
By the standard call***?Name*** it is possible to obtain*information* on all symbols
with the given***Name*** that have been generated in*modules* or by the**Unique**
function as illustrates the following very simple fragment, namely:

In[2699]:= **n = 1; Clear[x, y, z]; While[n < 5, Unique[{x, y, z}]; n++]** In[2700]:= **?x***

▼Global `
x$4602
▼AladjevProcedures`
x x$ x$6012 x$6013 x$6014 x$6015 Thus, the names generated by the module behave in the same way, as other names concerning calculations. However, these names have the temporary character which defines, that they have to be completely removed from the system in the absence of need for them. Therefore, the majority of the*names* generated in modules will be removed after performance of these modules. Only names returned by modules explicitly remain. Moreover, outside of modules their local variables remain undefinite even if in the modules they received initial values. Meanwhile, it must be kept in mind that usage of the names of the form***name$nnn*** is the agreement of the**Mathematica** for the local variables generated by modules. Thus, in order to avoid any conflict situations with names of the specified form the user isn't recommended to use names of such format in own programs. It must be kept in mind that the variables generated by modules are unique only during the current session. The mechanism of use of*local* variables at the call of a module is considered rather in details in [30]. The local variables of modules allow assignment to them of initial values in the form of any expressions, including expressions, whose values can depend on the actual arguments

received at a module call or from external variables, for example**:**
In[2943]**:= G[x_, y_] := Module[**{**a = If[PrimeQ[x], NextPrime[y], If[PrimeQ[y], NextPrime[x], z]]**}**, a*(x + y)]** In[2944]**:= z = 90;** {**G[7, 500], G[6, 18], G[72, 67]**}
Out[2944]= {255021**,** 2 160**,** 10 147}
Meanwhile, at the level of the local variables not exists of any opportunity immediate*(without execution of offers of the body of a procedure)* of an exit from the procedure, for example, in case of calculation of the initial expressions ascribed to the local variables as it illustrates simple enough fragment, the exception is the use of the call**Abort[]** that initiates return by the procedure the value**$Aborted:**
In[2487]**:= G[x_, y_] := Module[**{**a = If[PrimeQ[x], NextPrime[y], Return[x]]**}**, a*(x + y)]**
In[2488]**:= G[90, 500]**
Out[2488]= 53 100
In[2489]**:= G[x_, y_] := Module[**{**a = If[PrimeQ[x], NextPrime[y], Defer[G[x]]]**}**, a*(x + y)]**
In[2490]**:= G[90, 500]**

Out[2490]= 590 G[90]

In[2491] **:= G[x_, y_] := Module[**{**a = If[PrimeQ[x], NextPrime[y], Abort[]]**}**, a*(x + y)]**
In[2492]**:= G[90, 500]**
Out[2492]= **$**Aborted

The concept of ***modules*** in the context of the mechanism of local variables quite closely is adjoined the objects of***block*** type, whose organization has the following kind, namely**:**

Block[{***a, b, c,…***}**,*Body*]– *Body*** is evaluated, using local values for variables {***a, b, c, …***}**;**
Block[{***a = a0, b = b0, c = c0,…***}**,*Body*]– *Body*** at initial values for variables {***a, b, c,…***} localized in the block is evaluated.

In modular structure local variables are such by definition whereas in block structure the variables, defined as local, operates only within the block. At that, if to them in the block aren**’**t ascribed values, they accept values of the variables of the same name that are external with respect to the block, while in case of assignment of values by it in the block, values of the variables of the same name outside of the block remain without change. Therefore, by this circumstance the mechanisms of local variables of modules and blocks enough significantly differ.
For this reason, speaking about procedures of types {***“Module”, “Block”***}**,** we have to analyze block objects regarding character of their local variables, namely**:** lack of local variables or existence for each local variable of initial value says that this object can be considered as the procedure that receives at a call an information only from the actual arguments. On such principle our tests considered here and in our books [30-32] for check of a block to be as an actual procedure are built. Thus, the general rule for a block structure is defined by the principle**–** the variables located outside of a block, until the block and after the block, save the values, acting in relation to the block as***global*** variables whereas in the block the variables of the same name can arbitrarily change the values according to the demanded algorithm. As it was noted above, the local variable***b*** in a

structure**Module[**{***b***}**,*Body*]** correlates with a unique symbol that is modified every time when the given module is used**;** the given symbol differs from the global name***b.*** Whereas the variable***b*** in a structure**Block[**{***b***}**,*Body*]** is global variable outside of the block, it in the course of performance of the block can accept any values but by exit from the block restores a value that it had on entrance to the block. If in case of a***Module*–**construction the*local* variables in such procedure body are especially temporary, in a***Block*–**construction they aren**'**t considered as such. Without going into detail, we only will note that for providing of the robustness of procedures it is recommended to program them, generally, on the basis of***Module***constructions, or on the basis of***Block***constructions for which there are no local variables or all local variables have initial values.

Meanwhile, it should be noted, within of modular structure a mechanism of localization of global variables can be quite realized similarly to mechanism used in a block structure**;** two variants have been offered in [32]. Moreover, also other variants of realization of the mechanism of localization of global variables that are used by blocks, and on the basis of other approaches one of which is considered in [30] are possible. In this regard quite appropriate to note, that there is quite simple and universal mechanism of work with global variables in the body of procedures keeping their values at the time of an entrance to procedure and of an exit from it. Formally its essence can be presented visually on the basis of the following rather simple scheme**:**

h*= *x*; *G*[*x*_, *y*_, …] := Module[**{b* = *h*, …***}**,*Body*<*h*, …,** {***h* = *b*, *Res***}**[[–1]]>]**

Suppose, outside of the body of a procedure ***G*** the*global* variable***h,*** used in the procedure as a*local* variable, received a value***x.*** The local variable***b*** of the procedure***G*** receives***x*** as an initial value, keeping it upto each potential exit from the procedure. In the future, the algorithm realized by procedure body can use the variable***h*** arbitrarily, and only each possible exit from the procedure***G*** along with return of a result*(**Res**)* has to provide assignment to variable***h*** of its initial value***x*** upto exit from the procedure. A quite simple example rather visually illustrates the described mechanism of use of global variables in a procedure body at the local level, namely**:**

In[2517] **:= h = 90; P[x_] := Module[**{**a = h, R**}**, h = 6; R = h*x;** {**h = a, R**}**[[–1]]]**
In[2518]**:=** {**P[500], h**}
Out[2518]= {3000**,** 90}
Thus, the block constructions allow to determine effectively**"***environment***"** in which it is temporarily possible to change values of global variables**;** global variables are used by a block body as local variables, and only on exit from the block they restore own values upto entrance to the block. In the general understanding, the block structures serve as certain***"fields"*** of the current session in which the values changes of the variables located outside of these areas without change of their values outside of such areas is admissible, i.e. some kind of*localization* of global variables of the current session in certain fields of computing space in general is provided. The given possibility was used rather effectively in a number of means composing our***AVZ_Package*** package [48]**.** At that, the block structure is implicitly used in realization of a number of the**Mathematica** functions, e.g.**Do, Table, Product, Sum,** etc., mainly, of iterative type for localization of variables of indexing as visually illustrates a very simple example, namely**:**
In[2520]**:= n := 90;** {{**Sum[n^2,** {**n, 18**}**], n**}**,** {**Product[n,** {**n, 25**}**], n**}} Out[2520]=

{{2109, 90}, {15511 210 043 330 985 984 000 000, 90}} As a rule, the system considers any variable, determined by the user in the current session if the opposite isn’t specified as a global variable. However, in certain cases is required to localize a global variable for some period, that the block structure quite successfully allows to make. At the same time it should be noted once again, the variables localized in a block only then are нoy local variables if in the block the values are ascribed to them; otherwise their values in the block will*coincide* with values of the variables of the same name that are external relative to the block. Thus, if for a localized variable in a block an initial value wasn’t defined, real localization for such variable isn’t made. Thus, in certain cases it makes sense to create procedural objects on a basis of both the modular, and the block organization. Therefore, in a general sense under*procedural* objects in the**Mathematica** system it is quite possible to consider the objects, created as on the basis of the modular, and block organizations on condition of fulfillment by the block organization of the mentioned conditions– absence for it of the local variables or existence for each local variable of an initial value. We will represent some means of manipulation with local variables of modules and blocks playing essential enough part in various problems of procedural programming. First of all, it is very desirable to have the means for determination of local variables of blocks and modules which are absent among standard means of the**Mathematica** system. As the similar means, it is possible to consider **Locals** procedure whose source code with examples of usage the following fragment represents. This procedure is intended for evaluation of the local variables of the user blocks and modules.

```
In[2538] := M[x_, y_] := Module[{a = 90, b = 500}, (x + y)*(a + b)]; M[x_] := x;
M[x_/; IntegerQ[x]] := Block[{}, x^2]; M[x_Integer, y_ /; ListQ[y]] := Block[{a, b =
90}, x^2];

P[x__] := Module[{a = h, R}, h = 90; R = h*x; {h = a, R}[[–1]]]

In[2539] := Locals[x_ /; BlockFuncModQ[x], R___] := Module[{c, d = {}, p, t, a =
Flatten[{PureDefinition[x]}], b = Flatten[{HeadPF[x]}], k = 1, Sg}, Sg[y_String] :=
Module[{h = 1, v = {}, j, s = ””, z = StringLength[y]–1}, Label[avz]; For[j = h, j <= z,
j++, s = s <> StringTake[y, {j, j}]; If[! SameQ[Quiet[ToExpression[s]], $Failed] &&
StringTake[y, {j + 1, j + 1}] == “,”, AppendTo[v, s]; h = j + 3; s = ””; Goto[avz]]];
AppendTo[v, s <> StringTake[y, {–1,–1}]]; Map[If[Quiet[StringTake[#, {1, 2}]] ===
“, “, StringTake[#, {3,–1}], #] &, v]]; c = Flatten[Map[Map3[StringJoin, #, {” := “, ”
= “}] &, b]]; c = Map[StringReplace[#, Map[Rule[#, ””] &, c]] &, a]; For[k, k <=
Length[a], k++, p = c[[k]]; If[SuffPref[p, “Module[{“, 1], t = 8,

If[SuffPref[p, “Block[{“, 1], t = 7, t = 0; AppendTo[d, “Function”]]]; If[t != 0,
AppendTo[d, SubStrSymbolParity1[StringTake[p, {t,–1}], “{“, “}“][[1]]]];
Continue[]]; d = Map[StringReplace[#, {”{”–> “{$$90$$”, “, “–> “, $$90$$”,

“= ” –> “= $$90$$”}] &, d]; d = Map[If[MemberQ[{“Function”, “{}”}, #], #,
Sg[StringTake[#, {2,–2}]]] &, d]; d = Map[If[FreeQ[Quiet[ToExpression[#]],

$Failed], #, StringJoin1[#]] &, d]; d = Map[If[# === {””}, “{}“, #] &,

Mapp[StringReplace, d, {“$$90$$”–> ””, “\”–> ””}]]; Map[Remove,
Names[“`$$90$$*”]]; d = If[Length[d] == 1, Flatten[d], d]; If[{R}!= {}&& !
HowAct[R], If[d === {”{}”}, R = {}, {b, k, R}= {d, 1, {}}; While[k <= Length[b], p =
b[[k]]; AppendTo[R, If[MemberQ[{”{}“,
“Function” }, p], p, If[StringQ[p], If[StringFreeQ[p, ” = “], {p, “No”}, StringSplit[p,
” = “]], Map[If[StringFreeQ[#, ” = “], {#, “No”}, StringSplit[#, ” = “]] &, p]]]]; k++];
R= If[NestListQ[R] && Length[R]==1, R[[1]], R]]]; If[d ==={”{}”}, {}, d]]
In[2540] := Locals[M]
Out[2540]= {{”a“, “b= 90”}, {”a= 90“, “b= 500”}, “{}“, “Function”} In[2541]:=
Locals[P]
Out[2541]= {”a= h“, “R”}
In[2542]:= Map[Locals, {ModuleQ, Definition2, Locals}]
Out[2542]= {{”a= PureDefinition[x]“, “b”}, {”a“, “b= Attributes[x]“, “c”},
{”c“, “d= {}“, “p“, “t“, “a= Flatten[{PureDefinition[x]}]“, “b= Flatten[{HeadPF[x]}]“,
“k= 1“, “Sg”}}
In[2543]: = G[x_, y_] := Module[{a = If[PrimeQ[x], NextPrime[y], If[PrimeQ[y],
NextPrime[x], z]]}, a*(x + y)]; Locals[G]
Out[2543]= {”a= If[PrimeQ[x], NextPrime[y], If[PrimeQ[y],
NextPrime[x], z]]”}
In[2544]:= Locals[P, t]
Out[2544]= {”a= h“, “R”}
In[2545]:= t
Out[2545]= {{”a“, “h”}, {”R“, “No”}}
In[2546]:= Locals[M, t1]
Out[2546]= {{”a“, “b= 90”}, {”a= 90“, “b= 500”}, “{}“, “Function”}
In[2547]:= t1
Out[2547]= {{{”a“, “No”},{”b“, “90”}}, {{”a“, “90”},{”b“, “500”}}, “{}“,
“Function”}
In[2548]:= Z[x_] := x; Z[x_, y_] := x + y; Z[x_, y_, z_] := x + y + z; Locals[Z, t2]
Out[2548]= {”Function“, “Function“, “Function”}
In[2549]:= t2
Out[2549]= {”Function“, “Function“, “Function”}
In[2550]:= Locals[G, t3]
Out[2550]= {”a= If[PrimeQ[x], NextPrime[y],
If[PrimeQ[y], NextPrime[x], z]]”}
In[2551]:= t3
Out[2551]= {”a“, “If[PrimeQ[x], NextPrime[y],
If[PrimeQ[y], NextPrime[x], z]]”}
In[2552]:= B[x_] := Module[{a}, x]; {Locals[B, t4], t4}
Out[2552]= {{”a”}, {”a“, “No”}}
In[2553]:= V[x_] := Module[{a, b, c, d}, x]; Locals[V, t5]
Out[2553]= {”a“, “b“, “c“, “d”}
In[2554]:= t5
Out[2554]= {{”a“, “No”}, {”b“, “No”}, {”c“, “No”}, {”d“, “No”}}
In[2555]:= B1[x_] := Module[{}, x]; {Locals[B1, t6], t6}
Out[2555]= {{}, {}}
```

The presented **Locals** procedure on functionality covers both the procedure of the same name, and the**Locals1** procedure that are considered in [30-32]. The procedure call**Locals[*x*]** returns the list whose elements in string format represent local variables of a block or a module***x*** together with their initial values. While the call**Locals[*x, y*]** with the second optional argument***y*** **–** an undefinite variable**–** provides in addition return through***y*** or of the simple ***or of the simple*** element list, or the list of***ListList***type with***type with***element sublists whose*first* elements determine names of local variables of a block/module***x*** in string format, whereas the second element**–** the initial values, ascribed to them in string format**;** absence of initial values is defined by the***"No"*** symbol. If an object***x*** has no local variables, the procedure call**Locals[*x, y*]** returns empty list, i.e. {}, the same result is returned thru the second optional argument***y*.** Moreover, on typical function the call of the procedure returns***"Function".*** The procedure is widely used in problems of manipulation by*local variables***.**

In[2352]**:= StringJoin1[x_ /; ListQ[x] && DeleteDuplicates[Map[StringQ, x]] == {True}] := Module[{a = x, b = Length[x], c = "", k = 1}, While[k <= b–1, c = c <> a[[k]] <> ", "; k++]; c <> a[[–1]]]]**

In[2353] **:= StringJoin1[{"Avz", "Agn", "Vsv", "Art", "Kr"}]**
Out[2353]= **"**Avz**,** Agn**,** Vsv**,** Art**,** Kr**"**
In[2354]**:= StringJoin1[{"Avz", 90, x + y, "Art", Sin}]**
Out[2354]= StringJoin1[{"Avz"**,** 90**,** x+ y**,** **"**Art"**,** Sin}]
The definition of the**Locals** procedure along with standard means uses and our means such as**BlockFuncModQ, PureDefinition, SuffPref, HowAct, HeadPF, Mapp, Map3, NestListQ, SubStrSymbolParity1** considered in the present book and in [28-33]. At that, for realization of this procedure along with the mentioned means the expediency became clear to define a simple **StringJoin1** procedure for the purpose of special processing of strings lists which is a modification of the standard**StringJoin** function. The procedure call**StringJoin1[*x*]** returns result of consecutive concatenation of the string elements of a list***x*** that are separated by commas as very visually illustrates a example of the previous fragment with source code of this procedure. The **StringJoin1** procedure belongs to group of the means operating with string structures, however it is considered exactly here in the context of the**Locals** procedure; the given procedure has a rather wide range of appendices. Meanwhile, in certain cases it is required to define only the list of names of local variables irrespectively from the initial values ascribed to them. The **Locals1** procedure replacing***2*** former procedures**Locals1** and**Locals2** [30] can be used for this purpose. The call**Locals1[*x*]** returns the list of names in string format of local variables of a block or a module***x;*** in case of absence of local variables for an object***x*** the procedure call returns the*empty* list, i.e. {}. Furthermore, in case of an object of the same name***x*** which contains the subobjects of the same name with*different* headings a nested list is returned whose elements are bijective with subobjects***x*,** according to their order at application of the**PureDefinition** procedure to the object***x*.** The following fragment represents source code of the **Locals1** procedure along with the most typical examples of its usage.

In[2587] **:= Locals1[x_ /;BlockFuncModQ[x]] := Module[{a, b ={}, c, k =1, kr}, kr[y_List] := Module[{d = {}, v = Flatten[y], j = 1}, While[j <= Length[v]/2, AppendTo[d, v[[2*j–1]]]; j++]; d]; ClearAll[a]; Locals[x, a]; If[NestListQ1[a], For[k, k <= Length[a], k++, c = a[[k]]; AppendTo[b, If[MemberQ[{"{}", "Function"}, c], c,**

kr[c]]]];

If[StringQ[PureDefinition[x]], Flatten[b], b], kr[a]]] In[2588]:**= Locals1[P]**
Out[2588]= {"a", "R"}

In[2589] **:= Locals1[G]**
Out[2589]= {"a"}
In[2590]**:= Locals1[M]**
Out[2590]= {{"a", "b"}, {"a", "b"}, "{}", "Function"}

For examples of this fragment the means, given in the fragment above, that represents the**Locals** procedure, have been used. Right there we will note that the**Locals1** procedure represents one of the most effective means in the problems of processing of local variables of blocks and modules.

As one more useful means of this kind it is possible to note also the **Locals2** procedure that should be preceded by the procedure, in many cases useful and intended for testing of the objects of the same name which have several definitions of various type. The procedure call**QBlockMod[*x*]** returns*True* if definitions of an object***x*** have type {***Module,Block***}, and*False* otherwise. This procedure assumes that in the presence among definitions of***x*** of the definitions of other type, such object in general can't be considered by the object of type {***Module,Block***}. It allows to allocate from the objects of the same name the objects of type {***Module, Block***}. The fragment represents source code of the**QBlockMod** procedure with examples of its typical use.

In[2585] **:= M[x_, y_] := Module[{a = 90, b = 500}, (x + y)*(a + b)]; M[x_] := x; A[m_, n_] := Module[{a = 42.72, b = {m, n=90}}, h*(m+n+p)/(a+b)]; A[m_] := Block[{a = 42.72, b = {m, n = 90}, q, t}, h*(m+n+p)/(a+b)]**

In[2586] **:= QBlockMod[x_] := Module[{a = Flatten[{PureDefinition[x]}], b, c = True, k = 1}, If[MemberQ[{"System", $Failed}, a[[1]]], False, b = Flatten[{HeadPF[x]}]; While[k <= Length[a], If[! SuffPref[StringReplace[a[[k]], b[[k]] <> " := "–> ""], 1], {"Module[{", "Block[{"}, 1], c = False; Break[]]; k++]; c]]**

In[2587] **:= A[x_] := Block[{}, x^3]; QBlockMod[M]**
Out[2587]= False
In[2588]**:= Map[QBlockMod, {ProcQ, A, ToString1, StrStr, 500, Sin}]** Out[2588]= {True, True, True, False, False, False}

In the context of the previous **QBlockMod** procedure the following**Locals2** procedure on objects***x*** of the type {***Module, Block***} returns the nested list of their local variables in string format without initial values assigned to them if the object***x*** contains several definitions, otherwise simple list is returned. The following fragment represents the procedure code along with examples of its usage; these examples are tooken from the previous fragment.

In[2710] **:= Locals2[x_ /; QBlockMod[x]] := Module[{c={}, d, p, h={}, k=1, j=1, a = Flatten[{PureDefinition[x]}], b = Flatten[{HeadPF[x]}]}, While[k <= Length[a], AppendTo[c, d = StringReplace[a[[k]], Mapp[Rule, Map[b[[k]] <> " := " <> # &, {"Module[", "Block["}], ""], 1]; Quiet[SubStrSymbolParity[d, "{", "}", 1][[–1]]]]; k++]; While[j <= Length[c], p = c[[j]]; p = Map[ToString,**

```
ToExpression[StringReplace[p, {“, “–> “$$90$$, “, ” = “–> “$$90$$–> “}]]]; p =
If[Length[p] == 1, p[[1]], p]; p = Map[If[StringFreeQ[#, “–> “],
StringReplace[#,”$$90$$”–> ””], StringReplace[StringTake[#, {1,
Flatten[StringPosition[#, “–> “]][[1]]–1}], ””$$90$$”–> ””]] &, p]; AppendTo[h, p];
j++]; If[Length[h] == 1, h[[1]], h]] In[2711]:= Locals2[A]

Out[2711] = {{”a“, “b”}, {”a“, “b”}, {”a“, “b“, “q“, “t”}}
In[2712]:= Locals2[M]
Out[2712]= Locals2[M]
```

In a number of cases there is a need of dynamic extension of the list of local variables for a block/module that is activated in the current session, without change of the object code on the storage medium. The**ExpLocals** procedure presented by the following fragment solves the problem. So, the procedure call**ExpLocals[*x*, *y*]** returns the list of local variables in string format with the initial values ascribed to them on which local variables of an object***x*** are expanded. At that, generally speaking this list can be less than a list***y*** given at the procedure call*(or at all empty)* because the variables that are available in the object***x*** as formal arguments or local variables are excluded from it.

```
In[2757]:= ExpLocals[P_ /; ModuleQ[P] || BlockQ[P], L_ /; ListQ[L] &&
DeleteDuplicates[Map[StringQ, L]] == {True}] := Module[{a =
Flatten[{PureDefinition[P]}][[1]], b = Locals1[P], c = Args[P, 90], d, p, p1, h, Op =
Options[P], Atr = Attributes[P]},

Quiet[d = Map[If[StringFreeQ[#, {” = “, “=”}], #, StringSplit[#, {” = “, “=”}][[1]]] &,
L]; p = Locals[P]; h = MinusList1[d, Flatten[{b, c}]]; If[h == {}, Return[{}]]; h =
Flatten[Map[Position[d, #] &, h]]; d = Join[p, c = Map[L[[#]] &, h]];
ToExpression[“ClearAllAttributes[” <> ToString[P] <> “]”]; ClearAll[P];
ToExpression[StringReplace[a, ToString[p]–> ToString[d], 1]]]; If[Op != {},
SetOptions[P, Op]]; SetAttributes[P, Atr]; c]

In[2758]:= Avz[x_] := Module[{a = 90, b, c}, a + x^2];
SetAttributes[Avz, Protected]; Agn[x_] := Module[{}, {x}];

Z[x_ /; IntegerQ[x]] := Module[ {a, b, c, d}, {a, b, c, d}[[x]]] In[2759]:=
ExpLocals[Agn, {“x”, “a = c + d”, “b”, “Art = 25”, “Sv”, “Kr = 18”}] Out[2760]=
{”a= c+ d“, “b“, “Art= 25“, “Sv“, “Kr= 18”}
In[2761]:= Definition[Agn]
Out[2761]= Agn[x_]:= Module[{a= c+ d, b, Art= 25, Sv, Kr= 18}, {x}] In[2762]:=
ExpLocals[Avz, {“x”, “a = c+d”, “b”, “Art = 25”, “Sv”, “Kr = 18”}] Out[2762]=
{”Art= 24“, “Sv“, “Kr= 16”}
In[2763]:= Definition[Avz]
Out[2763]= Attributes[Avz]= {Protected}

Avz[x _]:= Module[{a= 90, b, c, Art= 25, Sv, Kr= 18}, a+ x^2] In[2764]:=
ExpLocals[Avz, {“x”, “a = c+d”, “b”, “Art = 25”, “Sv”, “Kr = 18”}] Out[2764]= {}
In[2765]:= ExpLocals[Z, {“m = 90”, “n = 500”, “p = 72”}]
Out[2765]= {”m= 90“, “n= 500“, “p= 72”}
In[2766]:= Definition[Z]
Out[2766]= Z[x_ /; IntegerQ[x]]:= Module[{a, b, c, d, m=90, n=500, p=72},
```

{a, b, c, d}[[x]]]

The above fragment contains source code of the **ExpLocals** procedure with examples of its application to very simple procedures**Agn, Avz** and**Z** for the purpose of extension of their list of local variables the part of which has initial values; at that, in the second procedure the list of local variables is the empty whereas for the first procedure there is a nonempty crossing of the joint list of formal arguments and local variables with the list of variables on which it is necessary to expand the list of local variables of the procedure. If the joint list coincides with a list***y***, the procedure call returns the empty list, i.e. {}, without changing an initial object***x*** in the current session. It must be kept in mind that elements of the list***y*** need to be coded in string format in order to avoid assignment of values to them of variables of the same name of the current session and/or calculations according to initial values ascribed to them. The result of a modification of an initial object***x*** preserves options and attributes of the initial object***x*.**

It is known that activation in the current session of a module or a block in the field of names of variables of the system adds all their local variables as illustrates a simple example, namely:

In[2582] **:= Mb[x_] := Block[{Art = 25, Kr = 18, Sv = 47}, x]; Mb[90];** In[2583]**:= B[x_] := Block[{Art1 = 25, Kr1 = 18, Sv1 = 47}, x]; B[500];** In[2584]**:= Names["`*"]** Out[2584]= {"Art", "Art1", "B", "Kr", "Kr1", "Mb", "Sv", "Sv1"} Therefore, economical use of the local variables is quite important problem. Meanwhile, in the course of programming of blocks/modules quite really emersion of so-called*excess local variables***.** The**RedundantLocals** procedure which is based on the**ProcBMQ** procedure, in a certain degree solves this problem. The following fragment represents source code of the procedure.

In[2571]**:= RedundantLocals[x_ /; BlockFuncModQ[x]] := Module[{a, b, c, p, g, k = 1, j, v, t = {}, z = ""},**

**{ a, b}= {PureDefinition[x], Locals1[x]}; If[StringQ[a],
If[b == {}, True, p = Map[#[[1]] &, StringPosition[a, {"}= ", "}:= "}]]; p = Select[p, ! MemberQ[{"{"}", " "}"}, StringTake[a, {#–2, #}]] &];**

**c = Map[Map3[StringJoin, #, {" := ", " = "}] &, b]; g = Select[b, StringFreeQ[a, Map3[StringJoin, #, {" := ", " = "}]] &]; While[k <= Length[p], v = p[[k]];
For[j = v, j >= 1, j—, z = StringTake[a, {j, j}] <> z;**

If[! SameQ[Quiet[ToExpression[z]], $Failed], AppendTo[t, z]]]; z = ""; k++]; t = MinusList[g, Flatten[Map[StrToList, t]]]; If[t =={}, t, p = Select[Map[" "<> # <>"[" &, t], ! StringFreeQ[a, #]&]; g ={}; For[k = 1, k <= Length[p], k++, v = StringPosition[a, p[[k]]]; v = Map[#[[2]] &, v]; z = StringTake[p[[k]], {2,–2}]; c = 1; For[j = c, j <= Length[v], j++,

For[b = v[[j]], b <= StringLength[a], b++, z = z <> StringTake[a, {b, b}]; If[! SameQ[Quiet[ToExpression[z]], $Failed], AppendTo[g, z]; c = j + 1; z = StringTake[p[[k]], {2,–2}]; Break[]]]]]; MinusList[t, Map[HeadName[#] &, Select[g, HeadingQ1[#] &]]]]]], "Object <" <> ToString[x] <> "> has multiple definitions"]]

In[2572] **:= Map[RedundantLocals, {ProcQ, Locals1, RedundantLocals}]** Out[2572]=

{{"h"}, {"a"}, {}}
In[2573]:= **Vsv[x_, y_] := Module[{a, b, c = 90, d, h}, b = 500;**

d[z_] := z^2 + z + 500; h[t_] := Module[{}, t]; d[c + b] + x + y + h[x*y]] In[2574]:= **RedundantLocals[Vsv]**
Out[2574]= {"a"}
In[2575]:= **Map[RedundantLocals, {ProcQ, Globals, RedundantLocals,**

Locals, Locals1}]
Out[2575]= {{"h"}, {}, {}, {}, {"a"}}
In[3384]:= **RedundantLocalsM[x_ /; BlockFuncModQ[x]] := Module[{d, b = {}, k = 1, c = ToString[Unique["g"]], a = Flatten[{PureDefinition[x]}]},**

While[k <= Length[a], d = c <> ToString[x]; ToExpression[c <> a[[k]]]; AppendTo[b, If[QFunction[d], "Function", RedundantLocals[d]]]; ToExpression["ClearAll[" <> d <> "]"]; k++]; Remove[c]; If[Length[b] == 1, b[[1]], b]]

In[3385] := **Vsv[x_, y_] := Module[{a, b, c=90, d, h}, b=500; d[z_] := z*x+500; h[t_] := Module[{}, t]; d[c+b]+x+y+h[x*y]]; Vsv[x_, y_, z_] := Module[{a, b, c=90, d, h}, a=6; d[p_] := z+p+500; h[t_] := Module[{}, t]; d[c+b]+x+y+h[x*y*z]]; Vsv[x_] := x**
In[3386]:= **RedundantLocalsM[Vsv]**

Out[3386] = {{"a"}, {"b"}, "Function"}
The procedure call**RedundantLocals[*x*]** returns the list of local variables in string format of a block or a module***x*** which the procedure considers excess variables in the context, what both the initial values, and the values in body of the object***x*** weren't ascribed to them, or these variables aren't names of the internal functions or modules/blocks defined in body of the object***x*.** At that, the local variables, used as an argument at the call of one or the other function in the body of an object***x****(in particular,it takes place for our***Locals1** *procedure that uses the call***Locals[*x*, *a*]***in which thru the***2***nd optional argument* ***a*** *–an undefinite variable–return of additional result is provided)* also can get in such list. We will note, that the given procedure like the previous**ProcBMQ** procedure is oriented on single objects, whose definitions are unique while on the objects of the same name the call prints the message*"Object<x>has multiple definitions"***.** Meanwhile, unlike the previous**ProcBMQ** procedure, the **RedundantLocals** procedure quite successfully processes also objects containing in their bodies the definitions of typical functions*(with headings)***,** modules and blocks. However, the considering of this moment as a certain shortcoming of the**ProcBMQ** procedure isn't quite competent, discussion of that can be found in our books [32,33].

The **RedundantLocalsM** procedure completes the previous fragment, this procedure expands the**RedundantLocals** onto the objects of the same name. The call**RedundantLocalsM[*x*]** on a single object {***Block, Module***} is similar to the call**RedundantLocals[*x*]** whereas on a traditional function*"Function"* is returned**;** on an object***x*** of the same name {*block,traditional function,module*} the list of results of application of the**RedundantLocals** to all subobjects of the object***x*** is returned. Meanwhile, results of calls of the above procedures **RedundantLocals** and**RedundantLocalsM** suggest the additional analysis of an object***x*** concerning the excess local variables, i.e. these procedures can be considered as rather effective means for the preliminary analysis of the blocks/modules regarding an exception of excess local variables. As shows our experience, in most important cases the results of use of the procedures **RedundantLocals** and**RedundantLocalsM** can be considered as ultimate, without demanding any additional researches of an analyzed object.

At that, in the **Mathematica** system the evaluation of definitions of modules and blocks containing the duplicated local variables is made quite correctly without initiation of any erroneous or special situations which arise only at the time of a call of a block and a module, initiating erroneous situation of ***Block::dup*** and***Module::dup*** respectively, with return of the block/module call unevaluated. Meanwhile, the mechanism of identification of a*duplicated* local variable***h*** isn’t clear because in the list of local variables in definition of the***Proc*** procedure*at first* the variables***a*** and***d*** are located as rather visually illustrates the following quite simple fragment. For the purpose of definition of the fact of duplication of local variables in definitions of objects like block or module that are activated in the current session, the procedure has been created, whose call**DuplicateLocalsQ[*x*]** returns*True* in case of existence in definition of a procedure***x*** of duplication of local variables, otherwise*False* is returned. At that, in case of return of*True* thru the***2nd*** optional argument ***y****–an undefinite variable–* simple list or list of***ListList****-type* is returned whose elements define names of the duplicated local variables with multiplicities of their entries into the list of local variables. The next fragment represents source code of the**DuplicateLocalsQ** procedure and examples of its usage.

In[2560] **:= Proc[x__] := Module[{a, y, d={x}, h, c, h, d, a=6, h=2, c=7, a}, a*d]**
In[2561]**:= Proc[90, 500]**
Module**::**dup**:** Duplicate local variable***h*** found in local variable specification

{a , y, d= {90, 500}, h, c, h, d, a= 6, h= 2, c= 7, a}>> Out[2561]= Module[{a, y, d= {90, 500}, h, c, h, d, a= 6, h= 2, c= 7, a}, a*d] In[2562]**:= Blok[x__] := Block[{a, y, d ={x}, h, c, h, d, a = 6, h =2, c = 7, a}, a*d]** In[2563]**:= Blok[90, 500]**
Block**::**dup**:** Duplicate local variable***h*** found in local variable specification

{a, y, d= {90, 500}, h, c, h, d, a= 6, h= 2, c= 7, a}>> Out[2563]= Block[{a, y, d= {90, 500}, h, c, h, d, a= 6, h= 2, c= 7, a}, a*d] In[2564]**:= DuplicateLocalsQ[P_ /; BlockModQ[P], y___] := Module[{a, b = Locals1[P]},**

If[b == {}, False, b = If[NestListQ[b], b[[1]], b]; a = Select[Gather2[b],#[[2]] > 1 &]; If[a == {}, False, If[{y}!= {}&& ! HowAct[y], y = a]; True]]] In[2565]**:= {DuplicateLocalsQ[Proc, y], y}**

Out[2565] = {True, {{”a“, 3}, {”d“, 2}, {”h“, 3}}}
In[2566]**:= B[x_, y_] := Module[{a = 6}, a*x*y]; {DuplicateLocalsQ[B, t], t}**
Out[2566]= {False, t}

In[2590] **:= Proc[x_] := Module[{a, y, d ={x}, h, c, h, d, a =6, h = 2, c = 7, a}, a*d]; Proc1[x_] := Module[{}, {x}]; Proc2[x_] := Module[{a, a = 500}, a*x]; Blok42[x_] := Block[{a, y, d = {x}, h, c, h, d, a=6, h=2, c=7, a}, a*d];**

In[2591] **:= DuplicateLocals[x_ /; BlockModQ[x]] := Module[{a = Locals1[x]}, a = Select[Map[{#, Count[a, #]}&, DeleteDuplicates[a]], #[[2]] > 1 &]; a=Sort[a, ToCharacterCode[#1[[1]]][[1]] < ToCharacterCode[#2[[1]]][[1]] &]; If[Length[a] > 1 || a == {}, a, a[[1]]]]**

In[2592]:= **DuplicateLocals[Proc]**
Out[2592]= {{"a", 3}, {"c", 2}, {"d", 2}, {"h", 3}}
In[2593]:= **DuplicateLocals[Proc1]**
Out[2593]= {}
In[2594]:= **DuplicateLocals[Proc2]**

Out[2594] = {"a", 2}
In[2595]:= **DuplicateLocals[Blok42]**
Out[2595]= {{"a", 3}, {"c", 2}, {"d", 2}, {"h", 3}}

The **DuplicateLocals** procedure completes the previous fragment, its call **DuplicateLocals[*x*]** returns the simple or the nested list the first element of a list or sublists of which defines a name in string format of a multiple local variable of a block/module***x*** while the second defines its multiplicity. In the absence of multiple local variables the empty list, i.e. {} is returned. In this regard a certain interest a procedure presents whose call**DelDuplLocals[*x*]** returns the*name* of a module/block***x*,** reducing its*local* variables of the same name to***1*** with activation of the updated definition***x*** in the current session. Whereas the call**DelDuplLocals[*x*, *y*]** with the second optional argument***y*** **–** *an undefinite variable***–** through***y*** returns the list of excess local variables. At that, first of all only*simple* local variables*(without initial values)* are reduced. This procedure well supplements the previous**DuplicateLocals** procedure. The next fragment represents source code of the**DelDuplLocals** procedure along with examples of its typical usage.

In[2657] **:= Ag[x_, y_] := Module[{a, b, b = 90, c, b = 73, c = 500, c, d}, (a*b + c*d)*(x + y)]; Av[x_] := Block[{a = 90, a = 500}, a*x];**
In[2658]:= **As[x_] := Module[{a, b, c, a, c}, x]**
In[2659]:= **SetAttributes[Ag, {Protected, Listable}]; Art[x_] := Module[{}, x]**
In[2660]:= **DelDuplLocals[x_ /; BlockModQ[x], y___] := Module[{b = {}, d, c = {}, a = Locals[x], p}, If[a =={}, x, d =Attributes[x]; ClearAttributes[x, d]; Map[If[StringFreeQ[#, "="], AppendTo[b, #], AppendTo[c, #]] &, a]; b = DeleteDuplicates[b]; c = DeleteDuplicates[Map[StringSplit[#, " =", 2]&, c], #1[[1]]== #2[[1]]&]; p = Map[#[[1]] &, c]; b = Select[b, ! MemberQ[p, #] &]; c = Map[StringJoin[#[[1]], " = ", #[[2]]] &, c]; b = StringRiffle[Join[b, c], ", "]; p = PureDefinition[x]; ToExpression[StringReplace[p, StringRiffle[a, ", "]–> b, 1]]; SetAttributes[x, d]; If[{y}!= {}&& ! HowAct[y], y = MinusList[a, StrToList[b]], Null]; x]]**

In[2661] **:= DelDuplLocals[Ag]**
Out[2661]= Ag
In[2662]:= **Definition[Ag]**
Out[2662]= Attributes[Ag]= {Listable, Protected}

Ag[x _, y_]:= Module[{a, d, b= 90, c= 500}, (a b+ c d) (x+ y)] In[2663]:= **DelDuplLocals[Av]**
Out[2663]= Av
In[2664]:= Definition[Av]
Out[2664]= Av[x_]:= Block[{a= 90}, a x]
In[2665]:= **DelDuplLocals[As]**
Out[2665]= As

```
In[2666]:= Definition[As]
Out[2666]= As[x_]:= Module[{a, b, c}, x]
In[2667]:= DelDuplLocals[Ag, t]
Out[2667]= Ag
In[2668]:= t
Out[2668]= {”b“, “c“, “b= 73“, “c”}
```

Procedure provides processing of the objects having single definitions, but it is easily generalized to the objects of the same name. The fragment below represents the procedure expanding the previous procedure on a case of the blocks and modules of the same name. The procedure call**DelDuplLocalsM** is completely analogous to a procedure call**DelDuplLocals[*x*, *y*].**

```
In[2717] := Ag[x_, y_] := Module[{a, b, b = 90, c, b = 73, c = 500, c, d}, (a*b + c*d)*(x
+ y)]; Ag[x_] := Block[{a, b, b = 90, c, c = 500}, a*b*c*x]
In[2718]:= Ag[x_, y_, z_] := Module[{a = 73, a, b, b = 90, c = 500, c}, a*x + b*y +
c*z]; SetAttributes[Ag, {Protected, Listable}]
In[2719]:= Definition[Ag]
Out[2719]= Attributes[Ag]= {Listable, Protected}
Ag[x_, y_]:= Module[{a, b, b=90, c, b=73, c=500, c, d}, (a b+ c d)(x+ y)] Ag[x_]:=
Block[{a, b, b= 90, c, c= 500}, a b c x]
Ag[x_, y_, z_]:= Module[{a= 73, a, b, b= 90, c= 500, c}, ax+ b y+ c z]
```

```
In[2720] := DelDuplLocalsM[x_ /; BlockModQ[x], y___] := Module[{b, d, p, a =
Flatten[{PureDefinition[x]}], h = ToString[x], c = {}, z = {}}, If[Length[a] == 1,
DelDuplLocals[x, y], If[{y}!= {}&& ! HowAct[y] || {y}=== {}, b = Attributes[x];
ClearAttributes[x, b], Return[Defer[DelDuplLocalsM[x, y]]]]; Map[{AppendTo[c, d
= ToString[Unique[“vgs”]]], ToExpression[StringReplace[#, h <> “[“–> d <> “[“,
1]]}&, a]; p = Map[PureDefinition[#] &, Map[{DelDuplLocals[#, y],
Quiet[AppendTo[z, y]], Clear[y]}[[1]] &, c]]; ToExpression[Map[StringReplace[#,
GenRules[Map[# <> “[” &, c], h <> “[“], 1] &, p]]; SetAttributes[x, b];
Map[Remove[#] &, c]; If[{y}!= {}, y = z, Null]; x]]
```

```
In[2721] := DelDuplLocalsM[Ag, w]
Out[2721]= Ag
In[2722]:= Definition[Ag]
Out[2722]= Attributes[Ag]= {Listable, Protected}

Ag[x _, y_]:= Module[{a, b, b=90, c, b=73, c=500, c, d}, (a b+ c d)(x+ y)] Ag[x_]:=
Block[{a, b, b= 90, c, c= 500}, a b c x]
Ag[x_, y_, z_]:= Module[{a= 73, a, b, b= 90, c= 500, c}, ax+ b y+ c z]

In[2623]:= w
Out[2623]= {{”b“, “c“, “b= 73“, “c”}, {”b“, “c”}, {”a“, “b“, “c”}}
```

The above tools play rather essential part at debugging modules and blocks of rather large size, allowing on the first stages to detect the duplicated*local* variables of the same name and provide their reducing to one.

6.7. Global variables of modules and blocks; the means of manipulation

by them in the*Mathematica*software

Concerning the ***Maple***procedures the***Mathematica***procedures have more limited opportunities both relative to the mechanism of the global variables, and on return of results of the performance. If in case of a***Maple***procedure an arbitrary variable has been declared in***global***–section of the description, or which didn’t receive values in the body of a procedure on the operator of assignment***“:=”,***or on the system***assign*** procedure*(upto release**Maple 11**)* is considered as a global variable, then in a***Mathematica***–procedure all those variables which are manifestly not defined as local variables are considered as the global variables. The following example rather visually illustrates the aforesaid, namely**:**

```
In[2678] := Sv[x_] := Module[{}, y := 72; z = 67; {y, z}]
In[2679]:= {y, z}= {42, 47}; {Sv[2015], y, z}
Out[2679]= {{72, 67}, 72, 67}
```

Therefore, any redefinition in a***Mathematica***procedure*(module or block)* of a*global* variable*automatically redefines* the variable of the same name outside of the procedure, what demands significantly bigger attentiveness for the purpose of prevention of possible special and undesirable situations than in a similar situation with the***Maple***procedures. Thus, the level of providing the robustness of software in the**Mathematica** at using of the procedures is represented to us a little lower of the mentioned level of the**Maple** system. It should be noted that**Mathematica** allows definition of global variables of the procedures by means of a quite simple reception of modification of the mechanism of testing of the actual arguments at the time of a procedure call as it quite visually illustrates the following very simple fragment, namely**:**

```
In[2543] := Art[x_ /; If[! IntegerQ[x], h = 90; True, h = 500; True], y_] := Module[{a
= 2015}, x + y + h + a]
In[2544]:= {Art[90, 500], Art[18.25, 500]}
Out[2544]= {3105, 2623.25}
In[2545]:= Kr[x_, y_] := Module[{a = If[IntegerQ[x], 90, 500]}, x + y + a]
In[2546]:= {Kr[15.5, 500], Kr[90, 500]}
Out[2546]= {1015.5, 680}

In[2547] := Sv[x_, y_] := Module[{a = If[IntegerQ[x] && PrimeQ[y], 90, 500]}, x + y
+ a]
In[2548]:= {Sv[90, 500], Kr[18, 555]}
Out[2548]= {1090, 663}
In[2549]:= H[x_: 90, y_, z_] := Module[{}, x + y + z]
In[2550]:= {H[220, 250, 540], H[220, 250], H[540]}
Out[2550]= {1010, 560, H[540]}
```

In a number of cases this mechanism is a rather useful while for the **Maple** the similar modification of the mechanism of testing of types of the factual arguments at the time of a procedures call is inadmissible. Naturally, the similar mechanism is allowed and for**Maple** when an algorithm defined in the form of a test*(Boolean function)* of arguments is coded not in the heading of a procedure, but it is defined by a separate type with its activation in the current session. In such case the standard format***x::test*** is used for a testing of a***x***argument. By natural manner we can define also the initial values of local variables in a point of a procedure call depending on received values of its actual arguments as illustrate two examples of the previous fragment. At last, if the**Maple** system doesn**'**t allow assignment of values*by default* to intermediate arguments of a procedure, the**Mathematica**system allows that rather significantly expands possibilities of programming of procedures**;**the last example of the previous fragment gives a certain illustration to the told.

Meanwhile, it must be kept in mind that at using of the mechanism of*return* through global variables the increased attentiveness is required in order to any conflict situations with the global variables of the same name outside of procedures does not arise. Since the procedures as a rule will be repeatedly used in various sessions, the return through global variables is inexpedient. However, to some extent this problem is solvable at using, in particular, of the special names whose probability of emergence in the current session is extremely small, for example, on the basis of the**Unique** function. In case of creation of certain procedures of our package***AVZ_Package*** [48] the similar approach for return of results through global variables was used. In certain cases this approach is quite effective.

Analogously to the case of local variables the question of determination of existence in a procedure of global variables represents undoubted interest**;** in the first case the problem is solved by the procedures**Locals** and**Locals1** considered above, in the second**–** by means of two procedures**Globals** and **Globals1.** So, as a natural addition to**Locals1** the procedure whose the call **Globals[*x*]** returns the list of*global* variables in string format of a procedure ***x*** acts. The next fragment represents source code of the**Globals** procedure along with the most typical examples of its usage.

In[2522]**:= Globals[P_ /; ProcBMQ[P]] := Module[{c, d = {}, p, g = {}, k = 1, b = ToString1[DefFunc[P]], a = If[P === ExprOfStr, {}, Sort[Locals1[P]]]},**

**If[a == {}, Return[{}], c = StringPosition[b, {" := ", " = "}][[2 ;;–1]]]; For[k, k <= Length[c], k++, p = c[[k]];
AppendTo[d, ExprOfStr[b, p[[1]],–1, {" ", ",", "\"", "!", "{"}]]]; For[k = 1, k <= Length[d], k++, p = d[[k]]; If[p != "$Failed" && p != " ", AppendTo[g, If[StringFreeQ[p, {"{", "}"}], p, StringSplit[StringReplace[p, {"{"–> "", "}"–> ""}]], ","]], Null]]; g = Flatten[g]; d = {}; For[k = 1, k <= Length[g], k++, p = g[[k]]; AppendTo[d, If[StringFreeQ[p, {"[", "]"}], p, StringTake[p, {1, Flatten[StringPosition[p, "["]][[1]]–1}]]]]; g = d; d = {}; For[k = 1, k <= Length[g], k++, p = g[[k]]; AppendTo[d, StringReplace[p, {","–> "", " "–> ""}]]]; d = Sort[Map[StringTrim, DeleteDuplicates[Flatten[d]]]]; Select[d, ! MemberQ[If[ListListQ[a], a[[1]], a], #] &]]**

In[2523]**:= Sv[x_, y_] := Module[{a, b = 90, c = 500}, a = (x^2 + y^2)/(b + c); {z, h}= {a, b}; t = z + h; t]; GS[x_] := Module[{a, b = 90, c = 42},**

Kr[y_] := Module[{}, y^2 + Sin[y]]; a = x^2; {z, h, p}= {a, b, 18}; t = z + h*Kr[18]–Cos[x + Kr[90]]; t]; Ar[x_] := Module[{a, b = 90, c = 42, Kr, z}, Kr[y_] := Module[{}, y^2 + Sin[y]]; a = x^2; {z, h, p}= {a, b, 18}; t = z + h*Kr[18]–Cos[x + Kr[90]]; t]

In[2524] **:= Map[Globals, {Locals1, Locals, Globals, ProcQ, ExprOfStr, GS, DefFunc, Sv, Ar}]**

Out[2524]= {{”d“, “j“, “v”},{”h“, “R“, “s“, “v“, “z”}, {},{}, {}, {”h“,”Kr“, “p“, “t“, “z”}, {}, {”h“, “t“, “z”}, {”h“, “p“, “t”}}

The definition of the **Globals** procedure along with standard tools uses and our means such as**ProcBMQ, DefFunc, ExprOfStr, Locals1, ToString1** and **ListListQ** considered in the present book and in [28-33]. It should be noted that the procedure call**Globals[*P*]** the objects names of a procedure body to which assignments by operators {***“:=”, “=”***} are made and which differ from local variables of a main procedure***P*** understands as the global variables. Therefore, the situation when a*local* variable of a subprocedure in a certain procedure***P*** can be defined by the call**Globals[*P*]** as a global variable as it visually illustrates an example of application of the**Globals** procedure to our procedures**Locals** and**Locals1** containing the nested subprocedures***Sg*** and***Kr*** respectively is quite possible. In that case some additional research is required, or the**Globals** can be expanded and to this case, interesting as a rather useful exercise. For the solution of the given problem, in particular, it is possible to use the procedures**Locals1** and**Globals** in combination with the**MinusList** procedure [48].

One of simple enough variants of the generalization of the **Globals***(based on the**Globals**)* onto case of the nested procedures can be presented by a quite simple procedure**Globals1,** whose call**Globals1[*x*]** returns the list of global variables in string format of a procedure***x;*** at that, actual argument***x*** can be as a procedure that isn’t containing in the body of subprocedures of various level of nesting, and a procedure containing such subprocedures. Also some others interesting and useful in practical programming, the approaches for the solution of this problem are possible. The following fragment represents source code of the**Globals1** procedure along with examples of its usage.

In[3116] **:= Globals1[P_ /; ProcQ[P]] := Module[{a = SubProcs[P], b, c, d ={}}, {b, c}= Map[Flatten, {Map[Locals1, a[[2]]], Map[Globals, a[[2]]]}]; MinusList[DeleteDuplicates[c], b]]**

In[3117] **:= Map[Globals1, {Locals1, Locals, Globals, ProcQ, ExprOfStr, GS, DefFunc, Sv, Ar}]**
Out[3117]= {{},{”R”},{},{},{},{”h“,Kr“,”p“,”t“,”z”},{},{”h“,”t“,”z”},{”h“,”p“,”t”}}
In[3118]**:= P[x_, y_] := Module[{a, b, P1, P2}, P1[z_, h_] := Module[{m, n}, T = z^2 + h^2; T]; P2[z_] := Module[{P3}, P3[h_] := Module[{}, Q = h^4; Q]; P3[z]]; V = x*P2[x] + P1[x, y] + P2[y]; V]; P1[x_] := Module[{}, {c, d}= {90, 500}; c*d + x]; Map[Globals1, {P, P1}]**

Out[3118]= {{”Q“, “T“, “V”}, {”c“, “d”}}
In[3119]**:= Sv[x_, y_] := Module[{a, b = 90, c = 500}, a = (x^2 + y^2)/(b + c); {z, h}= {a, b}; t = z + h; gs = t^2]; Globals1[Sv]** Out[3119]= {”gs“, “h“, “t“, “z”}

In[3120] **:= LocalsGlobals[x_ /; ProcQ[x]] := {Locals[x], Globals1[x]}; LocalsGlobals[Sv]**
Out[3120]= {{”a“, “b= 90“, “c= 500”}, {”gs“, “h“, “t“, “z”}}

In particular, the first example is presented for comparison of results of calls of the procedures**Globals** and**Globals1** on the same tuple of arguments**;** so, if in the first case as the global variables of subprocedures were defined, in the second such variables as global don’t act any more. The function call **LocalsGlobals[*x*]** returns the nested list, whose the first element– the list of *localvariables* with initial values if such variables exist in string format while the second element defines the list of global variables of a procedure*(block or module)x.* The**ExtrNames** procedure is presented as a quite useful means at working with procedures, its initial code with examples of application is represented by the following fragment, namely**:**

In[2720] **:= ExtrNames[x_ /; ProcQ[x]] := Module[{a=BlockToModule[x], b, c, d, f, p = {}, g, k = 1}, {f, a}= {ToString[Locals[x]], Locals1[x]}; {b, c}= {HeadPF[x], PureDefinition[x]}; g = StringReplace[c, {b <> " := Module["–> "", ToString[f] <> ", "–> ""}]; d = Map[If[ListQ[#], #[[1]], #] &, StringPosition[g, {" := ", " = "}]]; For[k, k <= Length[d], k++, AppendTo[p, ExtrName[g, d[[k]],–1]]]; p = Select[p, # != ""&]; {a, Complement[a, p], Complement[p, a]}]**

In[2721] **:= GS[x_] := Block[{a = 90, b, c}, b = 500; c = 6; x = a + b + c; x]** In[2722]**:= ExtrNames[GS]**
Out[2722]= {{"a", "b", "c"}, {"a"}, {"x"}}
In[2723]**:= ExtrNames[ProcQ]**
Out[2723]= {{"a", "atr", "b", "c", "d", "h"}, {"atr", "h"}, {}}
In[2724]**:= ExtrNames[ExtrNames]**
Out[2724]= {{"a", "f", "b", "c", "d", "p", "g", "k"}, {"a", "b", "c", "f", "k"}, {}}
In[2727]**:= Globals2[x_ /; ProcQ[x]||ModuleQ[x]||BlockQ[x]] :=**

ExtrNames[x][[3]]

In[2728] **:= GS[h_] := Module[{a = 90, b, c}, b = 500; c = 6; x = a +b + c; x + h]**
In[2729]**:= VG[h_] := Block[{a = 90, b, c}, b = 500; c = 6; x = a +b + c; y = h^2]**
In[2730]**:= Map[Globals2, {GS, VG, ProcQ, Tuples1, TestArgsTypes,**

LoadFile}]
Out[2730]= {{"x"},{"x", "y"},{},{"Res"},{"$TestArgsTypes"},{"$Load$Files$"}}

The procedure call **ExtrNames[*x*]** returns the nested***3***–element list, whose *first* element defines the list of all local variables of a procedure***x*** in string format, the*second* element defines the list of local variables of the procedure ***x*** in string format to which in the procedure body***x*** are ascribed the values whereas the*third* element defines the list of global variables to which in the procedure body***x*** are ascribed the values by operators {**":=",""="**}. A rather simple**Globals2** function completes the fragment, the function is based on the previous procedure and in a certain degree expands possibilities of the considered procedures**Globals** and**Globals1** onto procedures of any type**;** the function call**Globals2[*x*]** returns the list of names in string format of the global variables of a procedure***x.***

Nevertheless, the previous means which are correctly testing existence at a module/block of the global variables defined by assignments by operators {**":=", "="**}, aren't effective in cases when definitions of the tested modules/ blocks use assignments of type {***a, b, …***} {**=** |**:=**} {***a1, b1, …***} or***a*[[*k*]]** {**=** |**:=**}***b,*** simply ignoring them. The given defect is eliminated by the**LocalsGlobals1** procedure, whose call**LocalsGlobals1[*x*]** returns the nested***3***–element list whose*first* sublist contains names in string format of the*local* variables, the *second* sublist contains*local* variables with*initial values* in string format, and the*third* sublist–*global* variables in string format of a block/module***x.*** On argument***x*** of type different from*block*/*module*, a procedure call is returned unevaluated. The fragment below represents source code of the procedure **LocalsGlobals1** procedure along with typical examples of its usage.

In[2483] **:= LocalsGlobals1[x_ /; QBlockMod[x]] := Module[{c = ””, d, j, h ={}, k = 1, p, G, L, a = Flatten[{PureDefinition[x]}][[1]], b = Flatten[{HeadPF[x]}][[1]]}, b = StringReplace[a, {b <> ” := Module[“–> ””, b <> ” := Block[“–> ””}, 1]; While[k <= StringLength[b], d = StringTake[b, {k, k}]; c = c <> d; If[StringCount[c, “{“] == StringCount[c, “}“], Break[]]; k++]; b = StringReplace[b, c <> “,”–> ””, 1]; L = If[c == “{}“, {}, StrToList[StringTake[c, {2,–2}]]]; d = StringPosition[b, {” := “, ” = “}]; d = (#1[[1]]–1 &) /@ d; For[k = 1, k <= Length[d], k++, c = d[[k]]; p = ””; For[j = c, j >= 1, j—, p = StringTake[b, {j, j}] <> p; If[! Quiet[ToExpression[p]] === $Failed && StringTake[b, {j–1, j–1}] == ” “, AppendTo[h, p]; Break[]]]]; G = Flatten[(If[StringFreeQ[#1, “{“], #1, StrToList[StringTake[#1, {2,–2}]]] &) /@ (StringTake[#1, {1, Quiet[Check[Flatten[StringPosition[#1, “[“]][[1]], 0]]–1}] &) /@ h]; b = (If[StringFreeQ[#1, ” = “], #1, StringTake[#1, {1, Flatten[StringPosition[#1, ” = “]][[1]]–1}]] &) /@ L; d = DeleteDuplicates[Flatten[(StringSplit[#1, “, “] &) /@ MinusList[G, b]]]; d = Select[d, ! Quiet[SystemQ[#1]] && ! MemberQ[Flatten[{“\”, “#”, “"”, ””, “+”, “–“, ToString /@ Range[0, 9]}], StringTake[#1, {1, 1}]] &]; {Select[b, ! MemberQ[ToString /@ Range[0, 9], StringTake[#1, {1, 1}]] &], L, MinusList[d, b]}]**

In[2484]:= **M[x_, y_] := Module[{a = 90, b = 500, c = {v, r}}, h = x*y*a*b; m = 72; t := (a+b); g[[6]] = 73; t[z_] := a; {g, p}= {67, 72};**

{ k, j}:= {42, 72}; x+y]; {h, m, t, g, p, k, j}= {1, 2, 3, 4, 5, 6, 7}; {LocalsGlobals1[M], {h, m, t, g, p, k, j}} Out[2484]= {{{”a“, “b“, “c”}, {”a= 90“, “b= 500“, “c= {v, r}”}, {”h“, “m“, “t“, “g“, “p“, “k“, “j”}}, {1, 2, 3, 4, 5, 6, 7}}

In[2485] := **Sv[x_, y_] := Module[{a, b = 90, c = {n, m}}, a = (x^2 + y^2)/(b+c); {z, h}= {a, b}; t = z + h; gs = t^2]; LocalsGlobals1[Sv]**
Out[2485]= {{”a“, “b“, “c”}, {”a“, “b= 90“, “c= {n, m}”}, {”z“, “h“, “t“, “gs”}}
In[2486]:= **Vt[x_, y_] := Module[{a, b = 90, c ={n, m, {42, 72}}}, a =(x^2+y^2)/ (b + c); {z, h}= {a, b}; t = z + h; gs = t^2]; LocalsGlobals1[Vt]**
Out[2486]= {{”a“, “b“, “c”},{”a“, “b=90“, “c={n,m,{42,72}}”},{”z“, “h“, “t“, “gs”}}

In[2495]:= **LocalsGlobalsM[x_ /; QBlockMod[x]] := Module[{b = “$$90$”, c, d = {}, k = 1, a = Flatten[{PureDefinition[x]}]},**

While[k <= Length[a], c = b <> ToString[x]; ToExpression[b <> a[[k]]]; AppendTo[d, LocalsGlobals1[c]]; ToExpression[“Clear[” <> c <> “]”]; k++]; If[Length[d] == 1, d[[1]], d]]

In[2496]:= **M[x_, y_] := Module[{a = 90, b = 500, c}, h = x*y*a*b; m = 72; t := (a + b); g[[6]] = 73; t[z_] := a; {g, p}= {67, 72}; x + y];**

M[x_] := Block[{a, b}, y = x]; M[x__] := Block[{a, b}, {y, z}:= {a, b}]; P1[x_] := Module[{a, b = {90, 500}}, {c, d}= {p, q}; {a, b, h, g}= {42, 47, 67, 78}; c*d + x]
In[2497]:= **LocalsGlobalsM[M]**

Out[2497] = {{{”a“, “b“, “c”}, {”a=90“, “b=500“, “c”}, {”h“, “m“, “t“, “g“, “p“, “k“,

"j"}}, {{"a", "b"}, {"a", "b"}, {"y"}}, {{"a", "b"}, {"a", "b"}, {"y", "z"}}}
In[2498]:= **LocalsGlobalsM[P1]**
Out[2498]= {{"a", "b"}, {"a", "b= {90, 500}"}, {"c", "d", "h", "g"}}

Meanwhile, the **LocalsGlobals1** procedure correctly works only with the blocks/modules having unique definitions, i.e. with the objects other than objects of the same name. Whereas the**LocalsGlobalsM** procedure expands the**LocalsGlobals1** procedure onto case of the blocks/modules of the same name**;** the procedure call**LocalsGlobalsM[*x*]** returns the list of the nested ***returns the list of the nested*** element lists of the format similar to the format of results of return on the calls**LocalsGlobals1[*x*]** whose elements are biunique with subobjects of***x*,** according to their order at application to the object***x*** of the**PureDefinition** procedure. On arguments ***x*** of the type different from block/module, the procedure call**LocalsGlobalsM[*x*]** is returned unevaluated. Source code of the**LocalsGlobalsM** procedure along with examples of its use complete the previous fragment. Meanwhile, it must be kept in mind that the returned list of global variables doesn't contain multiple names though the identical names and can belong to objects of various type as very visually illustrates the first example to the**LocalsGlobals1** procedure in the previous fragment in which the symbol***"t"*** acts as a global variable twice. Indeed, the simple example below very visually illustrates the aforesaid, namely**:**

In[2538]**:= t[z_] := z; t := (a + b); Definition[t]**
Out[2538]= t**:=** (a+ b)
t[z_]**:=** z

Therefore carrying out the additional analysis regarding definition of types of the global variables used by the tested block/module in event of need is required. Definition of the**LocalsGlobals1** procedure along with standard means uses and our means such as**HeadPF, QBlockMod, PureDefinition, MinusList, StrToList, SystemQ**MinusList, StrToList, SystemQ 33]. The procedure has a number of applications at programming of various problems, first of all, of the system character.

Above we determined so -called*active* global variables as global variables to which in the objects of type {*Block*,*Module*} the assignments are done while we understand the global variables different from arguments as the***passive*** global variables, whose values are only used in objects of the specified type. In this regard means that allow to evaluate the passive global variables for the user blocks and modules are being represented as very interesting. One of similar means**–** the**BlockFuncModVars** procedure that solves even more general problem.The next fragment represents source code of the procedure **BlockFuncModVars** along with the most typical examples of its usage.

In[2337]**:= BlockFuncModVars[x_ /; BlockFuncModQ[x]] := Module[{d, t, c = Args[x, 90], a = If[QFunction[x], {}, LocalsGlobals1[x]], s ={"System"}, u = {"Users"}, b = Flatten[{PureDefinition[x]}][[1]], h ={}},**

d = ExtrVarsOfStr[b, 2]; If[a == {}, t = Map[If[Quiet[SystemQ[#]], AppendTo[s, #], If[BlockFuncModQ[#], AppendTo[u, #], AppendTo[h, #]]] &, d]; {s, u = Select[u, # != ToString[x] &], c, MinusList[d, Join[s, u, c, {ToString[x]}]]}, Map[If[Quiet[SystemQ[#]],AppendTo[s, #], If[BlockFuncModQ[#], AppendTo[u, #], AppendTo[h, #]]] &, d]; {Select[s, ! MemberQ[{"$Failed", "True", "False"}, #] &],

Select[u, # != ToString[x] && ! MemberQ[a[[1]], #] &], c, a[[1]], a[[3]], Select[h, ! MemberQ[Join[a[[1]], a[[3]], c, {"System", "Users"}], #] &]}]] In[2338]:= **A[m_, n_, p_ /; IntegerQ[p], h_ /; PrimeQ[h]] := Module[{a = 6}, h*(m+n+p)/a + StringLength[ToString1[z]]/(Cos[c] + Sin[d])]**

In[2339]:= **BlockFuncModVars[A]**
Out[2339]= {{"System", "Cos", "IntegerQ", "Module", "PrimeQ", "Sin",

" StringLength"}, {"Users", "ToString1"}, {"m", "n", "p", "h"}, {"a"}, {}, {"c", "d", "z"}}
In[2340]:= **BlockFuncModVars[StringReplaceS]**
Out[2340]= {{"System", "Append", "Characters", "If", "Length", "MemberQ",

"Module", "Quiet", "StringLength", "StringPosition", "StringQ", "StringReplacePart", "StringTake", "While"}, {"Users"},

{ "S", "s1", "s2"}, {"a", "b", "c", "k", "p", "L", "R"}, {}, {}} In[2341]:= **BlockFuncModVars[BlockFuncModVars]**
Out[2341]= {{"System", "AppendTo", "Flatten", "If", "Join", "MemberQ",

" Module", "Quiet", "Select", "ToString"}, {"Users", "Args", "BlockFuncModQ", "ExtrVarsOfStr", "LocalsGlobals1", "MinusList", "PureDefinition", "QFunction", "SystemQ"}, {"x"},

{ "d", "t", "c", "a", "s", "u", "b", "h"}, {}, {}}
In[2342]:= **BlockFuncModVars[LocalsGlobals1]**
Out[2342]= {{"System", "Append", "Block", "Break", "Check",

" DeleteDuplicates", "Flatten", "For", "If", "Length", "MemberQ", "Module", "Quiet", "Range", "Select", "StringCount", "StringFreeQ", "StringJoin", "StringLength", "StringPosition", "StringReplace", "StringSplit", "StringTake", "ToExpression", "ToString", "While"}, {"Users", "HeadPF", "MinusList", "PureDefinition", "QBlockMod", "StrToList", "SystemQ"},{"x"},{"c", "d", "j", "h", "k", "p", "G", "L", "a", "b"},{},{}}

In[2343]:= **BlockFuncModVars[StrStr]**
Out[2343]= {{"System", "If", "StringJoin", "StringQ", "ToString"}, {"Users"}, {"x"}, {}}

The procedure call **BlockFuncModVars[*x*]** returns the nested ***returns the nested***element list, whose*first* element– the list of the*system* functions used by a block/module ***x*,** whose first element is***"System"*** while other names are system functions in string format; the second element– the list of the user means used by the block/module***x*,** whose first element is***"Users"*** whereas the others define names of means in string format; the third element defines the list of formal arguments in string format of the block/module***x*;** the fourth element– the list of local variables in string format of the block/module***x*;**the fifth element – the list of active global variables in string format of the block/module***x*;** at last, the sixth element determines the list of*passive* global variables in string format of the block/module***x*.** While on a user function***x*** the procedure call **BlockFuncModVars[*x*]** returns the nested ***nested***element list, whose first element
– the list of the system functions used by a function***x*,** whose first element is ***"System"***

while other names are system functions in string format**;** the*2nd* element**–** the list of the user means used by the function***x*,** whose the first element is***"Users"*** whereas the others determine names of means in string format**;** the third element defines the list of formal arguments in the string format of the function***x;*** the fourth element**–** the list of global variables in string format of the function***x*.** The given procedure provides the structural analysis of the user blocks/functions/modules in the following contexts**:*(1)*** *the used system functions***,** ***(2)****the user means***,***active in the current session***,** ***(3)****the formal arguments***,** ***(4)****the local variables***,** ***(5)****the active global variables***,** and***(6)****the passive local variables*. The given means has a number of interesting enough appendices, first of all, of the system character.

The next procedure belongs to group of the means processing the strings, however, it is presented exactly here as it is very closely connected with the previous procedure; along with other our means it is the cornerstone of the algorithm of the **BlockFuncModVars** procedure. The following fragment represents source code of the**BlockFuncModVars** procedure along with the most typical examples of its usage.

```
In[2478] := ExtrVarsOfStr[S_/; StringQ[S], t_ /; MemberQ[{1, 2}, t], x___] :=
Module[{k, j, d = {}, p, a = StringLength[S], q = Map[ToString, Range[0, 9]], h = 1, c
= "", L = Characters["`!@#%^&*(){}:"\/|<>?~–=+[];:'., 1234567890"], R =
Characters["`!@#%^&*(){}:"\/|<>?~=[];:'., "]}, Label[G]; For[k = h, k <= a, k++, p
= StringTake[S, {k, k}]; If[! MemberQ[L, p], c = c <> p; j = k + 1; While[j <= a, p =
StringTake[S, {j, j}]; If[! MemberQ[R, p], c = c <> p, AppendTo[d, c]; h = j; c ="";
Goto[G]]; j++]]]; AppendTo[d, c]; d = Select[d, ! MemberQ[q, #] &]; d =
Select[Map[StringReplace[#, {"+"–> "", "–"–> "", "_"–> ""}] &, d], # != "" &]; d =
Flatten[Select[d, ! StringFreeQ[S, #] &]]; d = Flatten[Map[StringSplit[#, ", "] &, d]];

If[t == 1, Flatten, Sort][If[{x}!= {}, Flatten, DeleteDuplicates] [Select[d, !
MemberQ[{"\", "#", ""}, StringTake[#, {1, 1}]] &]]]]

In[2479] := A[m_, n_, p_ /; IntegerQ[p], h_ /; PrimeQ[h]] :=
Module[{a = 42.78}, h*(m + n + p)/a]
In[2480]:= ExtrVarsOfStr[Flatten[{PureDefinition[A]}][[1]], 2]
Out[2480]= {"a", "A", "h", "IntegerQ", "m", "Module", "n", "p", "PrimeQ"}
In[2481]:= G[x_, y_ /; IntegerQ[y]] := Module[{a, b = Sin[c + d], h}, z = x + y; V[m] +
Z[n]]
In[2482]:= ExtrVarsOfStr[PureDefinition[G], 1, 90]
Out[2482]= {"G", "x", "y", "IntegerQ", "y", "Module", "a", "b", "Sin", "c", "d", "h",
"z", "x", "y", "V", "m", "Z", "n"}
In[2483]:= V[x_, y_ /; PrimeQ[y]] := Block[{a, b = 73/90, c = m*n}, If[x > t + w, x*y,
S[x, y]]]
In[2484]:= ExtrVarsOfStr[PureDefinition[V], 2]
Out[2484]= {"a", "b", "Block", "c", "If", "m", "n", "PrimeQ", "S", "t", "V", "w", "x",
"y"}
In[2485]:= F[x_] := a*x + Sin[b*x] + StringLength[ToString1[x + c]];
BlockFuncModVars[F]
Out[2485]= {{"System", "Sin", "StringLength"}, {"Users", "ToString1"}, {"x"}, {"a",
"b", "c"}}
```

In[2486]:= **ExtrVarsOfStr["G[x_] := Module[{Vg, H73},
Vg[y_] := Module[{}, y^3]]", 1]**
Out[2486]= {"G", "x", "Module", "Vg", "H73", "y"}
In[2487]:= **ExtrVarsOfStr["(a + b)/(c + d) + Sin[c]*Cos[d + h]", 2]**
Out[2487]= {"a", "b", "c", "Cos", "d", "h", "Sin"}
In[2488]:= **ExtrVarsOfStr["(a + b)/(c + d) + Sin[c]*Cos[d + h]", 2, 90]**
Out[2488]= {"a", "b", "c", "c", "Cos", "d", "d", "h", "Sin"}

The procedure call **ExtrVarsOfStr[*S, t*]** at***t=2*** returns the sorted and at***t=1*** unsorted list of variables in string format, which managed to extract from a string***S;*** in the absence of similar variables the empty list, i.e. {} is returned. The procedure call**ExtrVarsOfStr[*S,t, x*]** with the*3rd* optional argument***x*** *– an arbitrary expression*– returns the list of the variables included in a string***S*** without reduction of their multiplicity to***1.*** Along with*standard* mechanism of local variables the**Mathematica** system allows use of mechanism of the global variables of the current session in the body of procedures as the local variables. Experience of use of the procedure confirms its high reliability in an extraction of variables; the procedure is quite simply adjusted onto the special situations arising in the course of its work. For correct application of the**ExtrVarsOfStr** procedure it is supposed that an expression***Exp,*** defined in a string***S*** is in the***InputForm***–format, i.e.***S* =ToString[InputForm[*Exp*]].** This procedure is effectively used at manipulations with definitions of the user blocks, functions, procedures. So, in the previous**BlockFuncModVars** procedure it is used very significantly.In general, procedure can be used for the analysis of algebraic expressions too.

A quite useful reception of ensuring use of the global variables which isn 't changing values of the variables of the same name outside of a procedure body was already given above. In addition to earlier described reception, we will present the procedure automating this process of converting at the time of performance of an arbitrary procedure of global variables to local variables of this procedure. The similar problem arises, in particular, in the case when it is required to execute a procedure having the global variables without changing their values outside of the procedure and without change of source code of the procedure in the current session. In other words, it is required to execute a procedure call with division of domains of definition of global variables of the current session of the system and global variables of the same name of the procedure. Whereas in other points of a procedure call such restrictions aren't imposed.

The **GlobalToLocal** procedure solves the task, whose call**GlobalToLocal[*x*]** provides converting of definition of a procedure***x*** into definition of the***$$$x*** procedure in which all*global* variables of the initial procedure***x*** are included into the tuple of*local* variables;the procedure call returns a procedure name activated in the current session which has no global variables. Whereas the call**GlobalToLocal[*x, y*]** with the*second* optional argument***y*** *–an undefinite variable*– in addition through it returns the nested list whose*first* element is sublist of local variables and the*second* element is sublist of global variables of a procedure***x.*** The procedure in a number of cases solves the problem of protection of variables, external in relation to a procedure***x.*** The following fragment represents source code of the**GlobalToLocal** procedure with the most typical examples of its usage.

In[2526]:= **GlobalToLocal[x_ /; QBlockMod[x], y___] := Module[{b, c, a = LocalsGlobals1[x]},**

If[Intersection[a[[1]], a[[3]]] == a[[3]] || a[[3]] == {}, x, b = Join[a[[2]], MinusList[a[[3]], a[[1]]]]; c = "$$$" <> StringReplace[PureDefinition[x], ToString[a[[2]]]–> ToString[b], 1]; If[{y}!= {}&& ! HowAct[y], y = {a[[1]], a[[3]]}]; ToExpression[c]; Symbol["$$$" <> ToString[x]]]]

In[2527]:= **GS[x_] := Module[{a, b = 90, c = {m, n}}, Kr[y_] := Module[{}, y^2 + Sin[y]]; a = x^2; {z, h, p}= {a, b, 5}; t = z + h*Kr[6]–Cos[x + Kr[9]]; t]**

In[2528] **:= GlobalToLocal[GS]** Out[2528]= $$$GS
In[2529]**:= Definition[$$$GS]** Out[2529]= $$$GS[x_]:= Module[{a, b= 90, c= {m, n}, Kr, z, h, p, t},

Kr[y _]:= Module[{}, y^2+ Sin[y]]; a= x^2; {z, h, p}= {a, b, 5}; t= z+ h Kr[6]– Cos[x+ Kr[9]]; t] In[2530]:= {**GlobalToLocal[GS, y], y**}
Out[2530]= {$$$GS, {{"a", "b", "c"}, {"Kr", "z", "h", "p", "t"}}}
In[2531]:= **LocalsGlobals1[$$$GS]**
Out[2531]= {{"a", "b", "c", "Kr", z", "h", "p", "t"}, {"a", "b= 90", "c= {m, n}",

"Kr", "z", "h","p", "t"}, {}}

The algorithm used by the **GlobalToLocal** procedure is rather simple and consists in the following. In case of absence for a procedure*x* of the global variables the name*x* is returned; otherwise, on the basis of definition of the *x* procedure, the definition of procedure with name*$$$x* which differs from the initial procedure only in that that the global variables of the*x* procedure are included into a tuple of local variables of the procedure*$$$x* is formed. Whereat, this definition is activated in the current session with return of the name*$$$x,* allowing to carry out the*$$$x* procedure in the current session without change of values of global variables of the current session.

At the same time the problem of converting of a block or a module *x* into an object of the same type in which*global* variables are included in tuple of the *local* variables of the returned object*x* of the same name with both the same attributes, and options is of interest. The**GlobalToLocalM** procedure solves this problem; the fragment below represents source code of the procedure **GlobalToLocalM** along with typical examples of its application.

In[2651] **:= GlobalToLocalM[x_ /; QBlockMod[x]] := Module[{d, h = "$$$", k = 1, n, p = {}, b = Attributes[x], c = Options[x], a = Flatten[{PureDefinition[x]}]}, While[k <= Length[a], d = a[[k]]; n = h <> ToString[x]; ToExpression[h <> d]; GlobalToLocal[Symbol[n]]; AppendTo[p, PureDefinition["$$$" <> n]]; ToExpression["ClearAll[" <> n <> "]"]; k++]; ClearAllAttributes[x]; ClearAll[x]; ToExpression[Map[StringReplace[#,"$$$$$$"–> "", 1] &, p]]; SetAttributes[x, b]; If[c != {}, SetOptions[x, c]];]**

In[2652] **:= A[x_] := Block[{}, g = x; {m, n}= {90, 6}]; A[x_, y_] := Module[{}, h = x + y; z = h*x]; SetAttributes[A, {Listable, Protected}]; GlobalToLocalM[A]**
In[2653]**:= Definition[A]**

Out[2653] = Attributes[A]= {Listable, Protected}

A[x_]:= Block[{g, m, n}, g= x; {m, n}= {6, 9}]
A[x_, y_]:= Module[{h, z}, h= x+ y; z= h*x]

In[2654]:= **GS[x_] := Module[{a, b = 500, c = {m, n}}, Kr[y_] := Module[{}, y^2 + Sin[y]]; a = x^2; {z, h, p}= {a, b, 18}; t = z + h*Kr[18]–Cos[x + Kr[90]]; t];**

GS[x_, y_] := Block[{a, b = {90, 500}, c}, z = x + y; d = Art]; SetAttributes[GS, {Protected}]; GlobalToLocalM[GS] In[2655]:= **Definition[GS]**
Out[2655]= Attributes[GS]= {Protected}

GS[x _]:= Module[{a, b= 500, c= {m, n}, Kr, z, h, p, t}, Kr[y_]:= Module[{}, y^2+ Sin[y]]; a= x^2; {z, h, p}= {a, b, 18}; t= z+ h*Kr[18]– Cos[x+ Kr[90]]; t]
GS[x_, y_]:= Block[{a, b= {90, 500}, c, z, d}, z= x+ y; d= Art]

In[2656] **:= GSV[x_] := Module[{a, b = 500, c = {m, n}}, Kr[y_] := Module[{}, y^2 + Sin[y]]; a = x^2; {z, h, p}= {a, b, 18}; t = z + h*Kr[18] + Cos[x + Kr[500]]; w = t^2]; GSV[x_, y_] := Block[{a, b ={90, 50}, c}, z = x +y; d =Art; t =Length[b]*z]; SetAttributes[GSV, {Protected}]; GlobalToLocalM[GSV]** In[2657]:= **Definition[GSV]**

Out[2657] = GSV[x_]:= Module[{a, b= 500, c= {m, n}, Kr, z, h, p, t, w}, Kr[y_]:= Module[{}, y^2+ Sin[y]]; a= x^2; {z, h, p}= {a, b, 18}; t= z+ hKr[18]+ Cos[x+ Kr[500]]; w= t^2]
GSV[x_, y_]:= Block[{a, b= {90, 500}, c, z, d, t}, z=x+ y; d=Art; t= Length[b]*z]

The procedure call **GlobalToLocalM[*x*]** returns*Null*, i.e. nothing, herewith converting a block or a module***x*** into the object***x*** of the same type and with the same attributes and options in which the*global* variables*(if they were)* of the initial object receive the*local* status. In the case of the objects of the same name***x*** the procedure call provides correct converting of all components of the object determined by various definitions. The fragment examples rather visually clarify the sense of similar converting.

It must be kept in mind, our package ***AVZ_Package*** [48] contains a number of other tools for the analysis of the procedures regarding existence in them of local and global variables, and also for manipulation with*arguments*,*local* and*global* variables of objects of the types {***Block, Function, Module***}. Means for work with*local* and*global* variables which are presented here and in [48] are quite useful in procedural programming in the**Mathematica** system.

Meantime, it must be kept in mind that a series of tools of the***AVZ_Package*** package can depend on a version of the**Mathematica** system, despite the a rather high level of*prolongation* of the builtin***Math***–language of the system. Therefore in some cases a certain tuning of separate means of the package onto the current version of the system can be demanded, what in principle for the rather experienced user shouldn't cause special difficulties. At that, similar tuning can be demanded even in case of passing from one operation platform onto another, for example, with***Windows XP*** onto***Windows 7.***

6.8. Attributes, options and values by default for the arguments of the user blocks, functions and modules; additional means of processing of them in*Mathematica*

The **Mathematica** system provides the possibility of assignment to variable, in particular, to names of blocks, functions or modules of the certain special attributes defining their different properties. So, the***Listable*** attribute for a function***W*** defines, that the function***W*** will be automatically applied to all elements of the list which acts as its argument. The current tenth version of the**Mathematica** system has***19*** attributes of various purpose**,**the work with them is supported by***3*** functions, namely**: Attributes, ClearAttributes** and **SetAttributes** whose formats are discussed, for example, in [30-33]. These***3*** functions provide such operations as**:*(1)*** return of the list of the attributes ascribed to an object***x;(2)*** deletion of all or separate attributes ascribed to an object***x; (3)*** a redefinition of the list of the attributes ascribed to an object***x.*** Meanwhile, in a number of cases of these means it isn**'**t enough or they are not so effective. Therefore, we offered a number of means in this direction which expand the above standard**Mathematica** means.

Above all**,**since eventually*new***Mathematica** versions quite can both change the standard set of attributes and to expand it, the problem of testing of an arbitrary symbol to be qua of an admissible attribute is quite natural. The **AttributesQ** procedure solves the given problem whose call**AttributesQ[*x*]** returns*True***,** if***x*** **–** the list of admissible attributes of the current version of the system, and*False* otherwise. Moreover, the call**AttributesQ[*x*, *y*]** with the***2nd*** optional argument***y*** **–***an undefinite variable***–** returns through it the list of elements of the list***x*** which aren**'**t attributes. The following fragment represents source code of the procedure with typical examples of its usage.

In[2550]:= **AttributesQ[x_List, y___] := Module[{a, b = {}}, Map[If[Quiet[Check[SetAttributes[a, #], $Failed]] === $Failed, AppendTo[b, #]] &, x]; If[b != {}, If[{y}!= {}&& ! HowAct[y], y = b]; False, True]]** In[2551]**:=** {**AttributesQ[**{**Listable, Agn, Protected, Kr, Art**}**, h], h**} Out[2551]= {False**,** {Agn**,** Kr**,** Art}}
In[2552]**:=** {**AttributesQ[**{**Protected, Listable, HoldAll**}**, g], g**} Out[2552]= {True**,** g} The given means is quite useful in a number of system appendices, at that, expanding the testing means of the**Mathematica** system.

Definitions of the user blocks, functions and modules in the **Mathematica** system allow qua of conditions and initial values for formal arguments, and initial values for local variables to use rather complex constructions as the following simple fragment illustrates, namely**:**

In[4173]**:= G[x_Integer, y_ /;** {**v[t_] := Module[**{}**, t^2], If[v[y] > 2015, True, False]**} **[[2]]] := Module[**{**a =** {**g[z_] := Module[**{}**, z^3], If[g[x] < 2015, 73, 90]**}**[[2]]**}**, Clear[v, g]; x*y + a]**

In[4174] **:=** {**a, b**}**=** {**500, 90**}**;** {**G[42, 73], G[42, 500], G[0, 0]**}
Out[4174]= {3156**,** 21090**,** G[0**,** 0]}
In[4175]**:= Map[PureDefinition,** {**v, g**}**]**
Out[2565]= {**"**v[t_]**:=** Module[{}**,** t^2]**"**, $Failed}

In[4176]**:= G[x_Integer, y_ /;** {**v[t_] := Module[**{}**, t^2], If[v[a] > 2015, True, False]**} **[[2]]] := Module[**{**a =** {**g[z_] := Module[**{}**, z^3],**

If[g[b] < 2015, 71, 90] }**[[2]]**}**, x*y + a]** In[4177]**:=** {**a, b**}**=** {**460, 71**}**;** {**G[42, 71], G[42, 460], G[0, 0]**}

```
Out[4177]= {3072, 19410, 90}
In[4178]:= Map[PureDefinition, {v, g}]
Out[4178]= {"v[t_]:= Module[{}, t^2]", "g[z_]:= Module[{}, z^3]"}
```
For possibility of use of sequence of offers, including as well definitions of procedures, as a condition for a formal argument*y* and initial value for the local variable*a* the reception that is based on the list has been used in the previous fragment. The sequences of offers were defined as elements of lists with value of their last element as*condition* and*initial value* respectively. At that, if in body of the main procedure***G*** a cleaning of symbols***v*** and***g*** from their definitions wasn't done, the procedures***v*** and***g*** will be available in the

current session, otherwise not. The given question is solved depending on an objective, the previous fragment illustrates the told. The above reception can be applied rather effectively for programming of means of the different purpose what illustrates a number of the procedures represented in [28-33]. The mechanisms of typification of formal arguments of the user functions, blocks and modules enough in details are considered in [30-33]. Meanwhile, along with the mechanism of typification of formal arguments, the system **Mathematica** has definition mechanisms for formal arguments of values by default, i.e. values that receive the corresponding factual arguments at their absence at the calls. However, the system mechanism of setting of values by default assumes definition of such values before evaluation of definitions of blocks**,**functions and modules on the basis of the standard**Default** function whose format supports installation of various values by default serially for separate formal arguments or of the same value for all arguments. The next fragment represents the procedure**Defaults1[*F, y*]** that provides the setting of expressions as values by default for the corresponding formal arguments of any subtuple of a tuple of formal arguments of the user block, function, module***F*** that is defined by the***2–***element list***y****(the first element–number of position of an argument,the second element is an expression)***.** For several values by default the list***y*** has***ListList***type whose sublists have the above format. The procedure successfully works with the user block, function or module ***F*** of the same name, processing only the first subobject from the list of the subobjects which are returned at the call**Definition[*F*].** The procedure call returns**$Failed,** or is returned unevaluated in special situations. The next fragment represents source code of the**Defaults** procedure with examples of its typical usage.

```
In[2640]:= Defaults[x_ /; BlockFuncModQ[x], y_ /; ListQ[y] && Length[y] == 2 || ListListQ[y] &&

DeleteDuplicates[Map[IntegerQ[#[[1]]] &, y]] == {True}] := Module[{a = Flatten[{Definition2[x]}], atr = Attributes[x], q, t, u, b = Flatten[{HeadPF[x]}][[1]], c = Args[x], d, p, h = {}, k = 1, g = If[ListListQ[y], y, {y}]}, If[Max[Map[#[[1]]&, y]] <= Length[c]&& Min[Map[#[[1]]&, y]] >= 1, c = Map[ToString, If[NestListQ[c], c[[1]], c]]; q = Map[#[[1]] &, y]; d = StringReplace[a[[1]], b–> "", 1]; While[k <= Length[q], p = c[[q[[k]]]]; t = StringSplit[p, "_"]; If[MemberQ[q, q[[k]]], u = If[Length[t] == 2, t[[2]] = StringReplace[t[[2]], " /; "–> ""]; If[Quiet[ToExpression["{" <> t[[1]] <> "=" <> ToString[y[[k]][[2]]] <> "," <> t[[2]] <> "}"]][[2]] || Quiet[Head[y[[k]][[2]]] === Symbol[t[[2]]]], True, False], True]; If[u, c[[q[[k]]]] = StringTake[p,

{1, Flatten[StringPosition[p, "_"]][[2]]}] <> "."]]; k++];
```

ClearAllAttributes[x]; ClearAll[x]; k = 1; While[k <= Length[q], ToExpression["Default[" <> ToString[x] <> ", " <> ToString[q[[k]]] <> "]" <> " = " <>

ToString1[y[[k]][[2]]]]; k++]; ToExpression[ToString[x] <> "[" <> StringTake[ToString[c], {2,–2}] <> "]" <> d]; Map[ToExpression, MinusList[a, {a[[1]]}]]; SetAttributes[x, atr], $Failed]]

```
In[2641] := G[x_, y_ /; IntegerQ[y]] := x+y; G[x_, y_, z_] := x*y*z; G[x_, y_, z_, h_]
:= x*y*z*h
In[2642]:= SetAttributes[G, {Listable, Protected, Flat}]
In[2643]:= Defaults[G, {{2, 500}, {1, 90}}]
In[2644]:= Definition[G]
Out[2644]= Attributes[G]= {Flat, Listable, Protected}
G[x_., y_.]:= x+ y
G[x_, y_, z_]:= x y z
G[x_, y_, z_, h_]:= x y z h
G /: Default[G, 1]= 90
G /: Default[G, 2]= 500
In[2645]:= {G[42, 47], G[73], G[]}
Out[2645]= {89, 573, 590}
In[2646]:= ClearAllAttributes[G]; ClearAll[G];
G[x_, y_ /; IntegerQ[y]] := x + y; G[x_, y_, z_] := x*y*z; G[x_, y_, z_, h_] := x*y*z*h
In[2647]:= SetAttributes[G, {Listable, Protected, Flat}]
In[2648]:= Defaults[G, {2, 90}]
In[2649]:= Definition[G]
Out[2649]= Attributes[G]= {Flat, Listable, Protected}
G[x_, y_.]:= x+ y
G[x_, y_, z_]:= x y z
G[x_, y_, z_, h_]:= x y z h
G /: Default[G, 2]= 90
In[2650]:= Defaults[G, {1, 500}]
In[2651]:= Definition[G]
Out[2651]= Attributes[G]= {Flat, Listable, Protected}
G[x_., y_.]:= x+ y
G[x_, y_, z_]:= x y z
G[x_, y_, z_, h_]:= x y z h
G /: Default[G, 1]= 500
G /: Default[G, 2]= 90
In[2652]:= {G[], G[72, 67], G[500]}
Out[2652]= {590, 139, 1000}
```

The successful call **Defaults[*G*, *y*]** returns*Null*, i.e. nothing, carrying out all settings***y*** of values by default for formal arguments of a block, function or module***G.*** It is necessary to emphasize once again that in case of an object ***G*** of the same name the call**Defaults[*G*,*y*]** processes only the first subobject from the list of subobjects which is returned on the call**Definition[*G*].** And this is a rather essential remark since the assignment mechanism to formal arguments of***G*** of values by default for case of an object of the same name, using the**Default** function, is other than ascribing for objects of such type, in particular, of the attributes. In the latter case the attributes are ascribed to all subobjects of an object***G*** of the same name whereas for values by default the mechanism is valid only concerning the*first* subobject from the list of the subobjects returned on the call**Definition[*G*].**This mechanism is realized as by the standard reception with use of the call**Default[*G*,*n*] =*default*** with template definition for***n***th formal argument of an object***G*** in the form***"_."*** and by the call**Defaults[*G*, {*n*, *default*}]** as it rather visually illustrates the following fragment, namely**:**

```
In[2673] := Clear[V]; Default[V, 2] = 90; V[x_, y_.] := {x, y};
V[x_, y_., z_, h_] := {x, y, z, h}
In[2674]:= Definition[V]
Out[2674]= V[x_, y_.]:= {x, y}
V[x_, y_., z_, h_]:= {x, y, z, h}
V /: Default[V, 2]= 90
In[2675]:= {V[500], V[42, 47, 67]}
Out[2675]= {{500, 90}, {42, 90, 47, 67}}
In[2676]:= Clear[V]; Default[V, 2] = 90; V[x_, y_.] := {x, y};
V[x_, y_, z_, h_] := {x, y, z, h} In[2677]:= Definition[V]
Out[2677]= V[x_, y_.]:= {x, y}
V[x_, y_, z_, h_]:= {x, y, z, h}
V /: Default[V, 2]= 90
In[2678]:= {V[500], V[42, 47, 67]}
Out[2678]= {{500, 90}, V[42, 47, 67]}
In[2679]:= Clear[V]; V[x_, y_] := {x, y}; V[x_, y_, z_, h_] := {x, y, z, h}; Defaults[V,
{2, 90}] In[2680]:= Definition[V]
Out[2680]= V[x_, y_.]:= {x, y}
V[x_, y_, z_, h_]:= {x, y, z, h}
V /: Default[V, 2]= 90
In[2681]:= {V[500], V[42, 47, 67]}
Out[2681]= {{500, 90}, V[42, 47, 67]}
```

While the **DefaultsM** procedure expands the previous**Defaults** procedure onto case of the objects of the same name of type {***Block***, ***Function***, ***Module***}. The successful procedure call**DefaultsM[*G*, *y*]** returns*Null*, i.e. nothing, at that, carrying out all settings of values by default***y*** for formal arguments of a block, function or module***G***. At that, for an object of the same name***G*** of the specified types the settings of values by default***y*** for formal arguments of all subobjects of the object***G*** are carried out. The next fragment represents source code of the**DefaultsM** procedure with typical examples of its usage.

```
In[2556]:= DefaultsM[x_ /; BlockFuncModQ[x], y_ /; ListQ[y] && Length[y] == 2 ||
ListListQ[y] &&

DeleteDuplicates[Map[IntegerQ[#[[1]]] &, y]] == {True}] := Module[{ArtKr, atr =
Attributes[x], q, k = 1,
a = Flatten[{PureDefinition[x]}], g = If[ListListQ[y], y, {y}]}, ClearAllAttributes[x];
ClearAll[x]; q = Map[#[[1]] &, g]; While[k <= Length[g], ToExpression["Default["
<> ToString[x] <> ", " <> ToString[g[[k]][[1]]] <> "]" <> " = " <> ToString1[g[[k]]
```

[[2]]]]; k++]; ArtKr[s_String, def_List] := Module[{n = Unique[AVZ], b, c, d, t, j= 1, h}, h = ToString[n] <> ToString[x]; ToExpression[ToString[n] <> s]; b = HeadPF[h]; d = StringReplace[PureDefinition[h], b–> ""]; c = Select[Map[ToString, Args[h]], # != "$Failed" &]; While[j <= Length[c], If[MemberQ[q, j], t = c[[j]]; c[[j]] = StringTake[t, {1, Flatten[StringPosition[t, "_"]][[2]]}] <> "."]; j++]; ToExpression[ToString[x] <> "[" <> StringTake[ToString[c], {2,–2}] <> "]" <> d]; ClearAll[h, n]]; k = 1; While[k <= Length[a], ArtKr[a[[k]], g]; k++]; SetAttributes[x, atr]] In[2557]:= **G[x_, y_, z_Integer] := x + y + z; G[x_, y_] := x + y; G[x_] := Block[{}, x]; G[x_, y_, z_, h_] := Module[{}, x*y*z*h]; SetAttributes[G, {Flat, Protected, Listable}];**

In[2558]:= **DefaultsM[G, {{2, 90}, {3, 500}}]**

In[2559]:= **Definition[G]**

Out[2559]= Attributes[G]= {Flat, Listable, Protected}

G[x_]:= Block[{}, x]

G[x_, y_.]:= x+ y

G[x_, y_., z_.]:= x+ y+ z

G[x_, y_., z_., h_]:= Module[{}, x y z h]

G /: Default[G, 2]= 90

G /: Default[G, 3]= 500

In[2550]:= {**G[56], G[42, 47], G[47, 18, 25]**}

Out[2550]= {56, 89, 90}

The**DefaultsM** procedure provides a rather useful expansion of standard means of this type, supporting as the single objects of type {***Block, Function, Module***}, and the objects of the same name as evidently illustrate examples of the previous fragment.

It is necessary to focus attention on one rather important point once again. As it was already noted earlier, the procedures can be defined on the basis of constructions of the types {***Module, Block***}. However, proceeding from certain considerations, it is generally recommended to give preference to the constructions of the***Module*** type because in a number of cases*(this question has been considered slightly above and in* [30-33]*in details)* the constructions of the***Block*** type are carried out incorrectly, without output of any diagnostic messages. As an illustration we will give an example of realization of the **Default1** procedure which concerns the theme of values by default, on the basis of two types of constructions– on the basis of**Module** and**Block.** The procedure call**Default1[*x, y, z*]** returns*Null*, i.e. nothing, providing settings of the values by default determined by a list***z*** for arguments of an object***x*,** whose positions are given by a list***y*** of***PosIntList***типа for a*block*/*function*/ *module****x*.** The next fragment from the standpoint of formalization represents almost identical realizations of definition of the**Default1** procedure on the basis of constructions and**Module,** and**Block.** And if the first realization is carried out quite correctly regardless of names of local variables, then the correctness of the second, generally speaking, depends on crossing of a list of names of local variables with a list of values by default for arguments, in particular, of a function as quite visually illustrates the following fragment in case when the local variable***a*** exists in addition and in the list of values by default for simple function***G*.** The following fragment represents source codes along with corresponding typical examples.

In[3792]:= **Default1[x_Symbol, y_ /; PosIntListQ[y], z_List] := Module[{k = 1, a = Min[Map[Length, {y, z}]]}, While[k <= a, Default[x, y[[k]]] = z[[k]]; k++];]**

In[3793]:= **Default1[G, {1, 2}, {a, b}]; G[x_., y_.] := {x, y};**
Clear[Default1]; DefaultValues[G]
Out[3793]= {HoldPattern[Default[G, 1]]:>a, HoldPattern[Default[G, 2]]:> b} In[3794]:=
Default1[x_Symbol, y_ /; PosIntListQ[y], z_List] := Block[{k = 1, a =
Min[Map[Length, {y, z}]]}, While[k <= a, Default[x, y[[k]]] = z[[k]]; k++];]
In[3795]:= **ClearAll[G]; Default1[G, {1, 2}, {a, b}]; G[x_., y_.] := {x, y};**
DefaultValues[G]
Out[3795]= {HoldPattern[Default[G, 1]]:>2, HoldPattern[Default[G, 2]]:> b} In[3796]:=
Default1[x_Symbol, y_ /; PosIntListQ[y], z_List] := Module[{k = 1, h =
Min[Map[Length, {y, z}]]}, While[k <= h, Default[x, y[[k]]] = z[[k]]; k++];]
In[3797]:= **Default1[G, {1, 2}, {a, b}]; G[x_., y_.] := {x, y};**
Clear[Default1]; DefaultValues[G]

Out[3797] = {HoldPattern[Default[G, 1]]:>a, HoldPattern[Default[G, 2]]:> b} In[3798]:=
Default1[x_Symbol, y_ /; PosIntListQ[y], z_List] := Block[{k = 1, h =
Min[Map[Length, {y, z}]]}, While[k <= h, Default[x, y[[k]]] = z[[k]]; k++];]
In[3799]:= **ClearAll[G]; Default1[G, {1, 2}, {a, b}]; G[x_., y_.] := {x, y};**

DefaultValues[G]
Out[3799]= {HoldPattern[Default[G, 1]]:>a, HoldPattern[Default[G, 2]]:> b}

Thus, the mechanisms of local variables used by procedures on the basis of **Module** and**Block,** generally, aren't identical. Consequently, in general it is necessary to give preference to definition of the procedures on the basis of a **Module** construction, however, taking into account the aforesaid there are very many cases when both types of the organization of the procedures are equivalent, demanding the preliminary analysis concerning the existence of such equivalence. The given question rather in details is considered in [33]. In general, for definition of the procedures we recommend to use structures on the basis of**Module** in order to avoid need of carrying out the additional analysis on procedurality and universality in all cases of appendices.

For determination of values by default for formal arguments of a function/ block/module it is possible to use both the means**Defaults, DefaultsM** and **Default,** and directly in their headings on the basis of constructions of the format***"x_:expression",*** or by combining both specified methods. However, the system**DefaultValues** function returns the settings of values by default, executed only by means of the standard**Default** function, for example**:**

In[2269] **:= Default[G5, 2] = 90; G5[x_, y_: 500, z_: 42] := {x, y, z};**
DefaultValues[G5]
Out[2269]= {HoldPattern[Default[G5, 2]]:> 90}
In[2270]:= **G5[Agn]**
Out[2270]= {Agn, 500, 42}
In[2271]:= **Default[S4, 2] = 90; S4[x_, y_., z_: 42] := {x, y, z};**
DefaultValues[S4]
Out[2271]= {HoldPattern[Default[S4, 2]]:> 90}
In[2272]:= **S4[Avz]**
Out[2272]= {Avz, 90, 42}

At that, if for argument a value by default has been defined and via**Default,** and directly in heading by a construction***"_:",*** then the second way has the maximum priority as it very visually illustrates the previous example with the***G5*** function. Meanwhile, the standard**DefaultValues** function possesses serious enough shortcomings. First of all, the given function doesn't reflect the values by default defined in a block/function/module heading, and only set through the**Default** function. However generally it is incorrect because for arguments the assignment of values by default as through the**Default** function, and directly in headings is admissible**;** at that, the priority belongs exactly to the second method what often can contradict result of a call of the **DefaultValues** function as it is illustrated with the previous examples.

For elimination of similar shortcomings the **DefaultValues1** procedure has been programmed, whose the call**DefaultValues1[*x*]** returns the list of the format {{***N1***}**:>*V1,* …,** {***Np***}**:>*Vp***}**,** where***Nj*** and***Vj(j=1..p)*** define numbers of positions of formal arguments in the heading of a block/function/module, and values by default ascribed to them respectively, regardless of method of their definition, taking into account the priority*(the setting of values by default in headings of blocks/functions/modules has the highest priority).* The following fragment represents source code of the**DefaultValues1** procedure with the most typical examples of its usage.

In[3079] **:= DefaultValues1[x_ /; BlockFuncModQ[x]]:= Module[{d ={}, h, k, a = {SetAttributes[String, Listable]}, b = Map[ToString, Args[x]], c = Map[ToString, DefaultValues[x]]}, ClearAttributes[ToString, Listable]; If[b != {}, For[a = 1, a <= Length[b], a++, h = b[[a]]; If[! StringFreeQ[h, "_:"],**

AppendTo[d, ToExpression[" {" <> ToString[a] <> "}:> " <> StringTake[h, {Flatten[StringPosition[h, "_:"]][[2]] + 1,–1}]]]]]]]; If[c != {}, If[c != {}, c = ToExpression[Mapp[StringReplace, Mapp[StringReplace, c, {"HoldPattern[Default[" <> ToString[x]–> "{", "]]"–> "}"}], {"{, "–> "{", "{}"–> "{2015}"}]]]; h = c[[1]][[1]]; If[Op[h] == {2015}, a = {}; For[k = 1, k <= Length[b], k++, AppendTo[a, ToExpression[ToString[{k}] <> " :> " <> ToString[c[[1]][[2]]]]]]; c = a]; If[PosIntListQ[h] && Length[h] > 1, a = {}; b = h; For[k = 1, k <= Length[b], k++, AppendTo[a, ToExpression[ToString[{k}] <> " :> " <> ToString[c[[1]][[2]]]]]]; c = a]]; If[d == {}&& c == {}, Return[{}],

c = Sort[Join[d, c], Op[#1][[1]][[1]] <= Op[#2][[1]][[1]] &]]; {k, h}= {1, {}}; While[k <= Length[c]–1, AppendTo[h, If[Op[c[[k]]][[1]] == Op[c[[k + 1]]][[1]], k + 1]]; k++]; Select[ReplacePart[c, Mapp[Rule, Select[h, # != "Null" &], Null]], ! SameQ[#, Null] &]]

In[3080] **:= Default[G] = 500; G[x_, y_., z_: 90] := {x, y, z}; DefaultValues1[G]**
Out[3080]= {{1}:> 500, {2}:> 500, {3}:> 90}
In[3081]**:= Default[S2, 2, 3] = 90; S2[x_, y_., z_] := {x, y, z}; DefaultValues1[S2]**
Out[3081]= {{1}:> 90, {2}:> 90}
In[3082]**:= Default1[S3, {1, 2, 3}, {42, 47, 25}]; S3[x_: 500, y_., z_.] := {x, y, z}; DefaultValues1[S3]**
Out[3082]= {{1}:> 500, {2}:> 47, {3}:> 25}

In[3083]:= **Default[S4, 2] = 2015; S4[x_: 500, y_: 47, z_: 42] :=** {**x, y, z**}**;**
DefaultValues1[S4]
Out[3083]= {{1}:> 500, {2}:> 47, {3}:> 42}
In[3084]:= **Default[S5, 2] = 90; S5[x_, y_: 500, z_: 42] :=** {**x, y, z**}**; DefaultValues1[S5]**
Out[3084]= {{2}:> 500, {3}:> 42}
In[3085]:= **Default1[V3,** {**1,2,3,4**}**,** {**a, b, c, d**}**]; V3[x_., y_., z_., t_.] :=** {**x,y,z,t**}**;**
DefaultValues1[V3]
Out[3085]= {{1}:> a, {2}:> b, {3}:> c, {4}:> d}
In[3086]:= **Default1[V4,** {**1, 2, 3, 4**}**,** {**a, b, c, d**}**]; V4[x_., y_: 90, z_., t_: 500] :=** {**x, y,**
z, t}**; DefaultValues1[V4]**
Out[3086]= {{1}:> a, {2}:> 90, {3}:> c, {4}:> 500}
Definition of the**DefaultValues1** procedure along with the standard means uses our means such as**Args, BlockFuncModQ, Mapp, PosIntListQ** and**Op** that are considered in the present book and in [28-33]. Thus, our procedure **DefaultValues1** rather significantly expands the possibilities of the standard ***DefaultValues*** function**,**and quite it replaces on condition of existence of the package***AVZ_Package*** [48] uploaded into the current**Mathematica** session.

Considering existence of ***2*** admissible mechanisms of assignment of values by default to formal arguments of the blocks, functions and modules, the problem of definition of this kind of values for objects of the specified type represents quite certain interest. In this connexion the**DefaultsQ** procedure solves the given problem whose call**DefaultsQ[*x*]** returns*True* if definitions of blocks, functions or modules***x*** contain values by default for their formal arguments, and*False* otherwise. Whereas the procedure call**DefaultsQ[*x*,*y*]** where the second argument***y*** *–an undefinite variable–* in addition through***y*** returns the list of the used types of values by default {**"_."**, **"_:"**}. The next fragment represents source code of the**DefaultsQ** procedure along with the most typical examples of its usage.

In[2776]:= **DefaultsQ[x_ /; BlockFuncModQ[x], y___] := Module[**{**c =** {}**, d, a = Args[x], b =** {**"_."**, **"_:"**}**, k = 1**}**, a = Map[ToString, If[NestListQ[a], a[[1]], a]];**

While[k <= Length[a], d = a[[k]]; If[! StringFreeQ[d, b[[1]]], AppendTo[c, b[[1]]], If[! StringFreeQ[d, b[[2]]], AppendTo[c, b[[2]]]]]; k++]; If[c == {}**, False, If[**{**y**}**!=** {}**&& ! HowAct[y], y = DeleteDuplicates[Flatten[c]]]; True]]** In[2777]:= **PureDefinition[G]**

Out[2777] = {"G[x_., y_.]:= x+ y", "G[x_, y_, z_]:= x*y*z",
"G[x_, y_, z_, h_]:= x*y*z*h"}
In[2778]:= {**DefaultsQ[G, t], t**}
Out[2778]= {True, {"_."}}
In[2779]:= **Default[S, 1] = 90; S[x_., y_: 500, z_] := x + y + z;**
Kr[x_, y_, z_] := Block[{}**, x*y*z]**
In[2780]:= {**Map9[DefaultsQ,** {**S, Kr**}**,** {**v1, v2**}**],** {**v1, v2**}}
Out[2780]= {{True, False}, {{"_.", "_:"}, v2}}
Along with attributes and values by default for formal arguments of a block, function or module, these objects can use the mechanism of***options.*** First of all, the mechanism of*options* is rather widely used by the system means. So, for a number of functions in the**Mathematica** system*(in particular, the**Plot** function)*, the options available both for

installation, and for redefinition are ascribed. The system supports the general mechanisms for work with such options. The call **Options[*G*]** returns the list of the current settings in the format {***a* –>*a1*,*b* –>*b1*, …**} for all options of a block, function or module***G*** while the call**Options[*G, h*]** returns the current setting for an option***h.*** In turn, the call**SetOptions[*G*,*a* –>*a2*,*b* –>*b2*, …]** provides a reinstalling of values for options {***a*,*b*,*c*, …**} of a block/function/module***G*** which remains active up to the next reinstalling in the current session. While the function call**SystemOptions[]** returns a list of the current settings for all preinstalled internal*options* and*suboptions* of the system. The given settings are defined as the used platform, and in certain cases also by the current session of the **Mathematica.** Thus, for receiving quantity of all system options and their quantities in the context as the groups of options, and the separate options, the following procedure**CountOptions** whose source code along with the most typical examples of its usage are given below is used, namely**:**

In[3252] **:= CountOptions[h___] := Module[**{**a = SystemOptions[], b =** {}**, d, c = 1, k**}**, While[c <= Length[a], d = a[[c]]; AppendTo[b, If[ListQ[Part[d, 2]],** {**Part[d, 1], Length[Part[d, 2]]**}**, d]]; c++]; b = Flatten[Gather[b, Head[#1] == Head[#2] &], 1]; If[**{**h**}**==** {}**, b, If[HowAct[h], Defer[CountOptions[h]], d = 0; Do[If[ListQ[b[[k]]], d = d + b[[k]][[2]], d = d + 1],** {**k, Length[b]**}**]];** {**h**}**=** {**d**}**; b]]** In[3253]**:= CountOptions[]**

Out[3253] = {{**”**AlgebraicsOptions**”**,8},{**”**AlgebraicThreadThroughHeads**”**,16}**, …, “**ZeroTestMaxPrecision**”–>** 5000**., “**ZeroTestNumericalPrecision**”–>** 80**.**} In[3254]**:= CountOptions[g]; g**
Out[3254]= 417

The call **CountOptions[]** returns the nested list whose elements are the lists and separate options. The list as the first element contains a name of group of options, whereas the second element**–** number of options in this group. While the call**CountOptions[*p*]** in addition thru argument***p*** ***–an undefinite variable–*** returns total of the preset system*options*/*suboptions*. Furthermore, settings for a concrete system option***p*** can be redefined by the function call **SetSystemOptions[*p* –>*value*],** however except for separate cases, it is not desirable in order to avoid the possible*conflicts* with the system settings. At that, the**Mathematica** system doesn**’**t support operations of removal of the options, therefore in the following fragment we present the**DeleteOptsAttr** procedure, decisive the given problem.

The procedure call **DeleteOptsAttr[*x*]** returns*Null***,** i.e. nothing, canceling for a symbol***x*** the options ascribed to it. While the call**DeleteOptsAttr[*x, y*],** returning*Null***,** i.e. nothing, cancels both the options, and the attributes that are ascribed to a symbol***x*,** where***y*** **–** an arbitrary expression. The following fragment represents source code of the**DeleteOptsAttr** procedure with the most typical examples of its usage.

In[2576] **:= G[x_, y_] := x^2 + y^2; Options[G] =** {**Art–> 25, Kr–> 18**} Out[2576]= {Art–> 25**,** Kr–> 18}
In[2577]**:= SetOptions[G, Art–> 25, Kr–> 18]**
Out[2577]= {Art–> 25**,** Kr–> 18}
In[2578]**:= SetAttributes[G,** {**Protected, Listable**}**]**
In[2579]**:= Definition2[G]**
Out[2579]= {**”**G[x**_**, y_]**:=** x^2+ y^2**“, “**Options[G]**:=** {Art**–>** 25**,** Kr**–>** 18}**“,**

{Listable, Protected}
In[2580]:= **DeleteOptsAttr[x_ /; BlockFuncModQ[x], y___] := Module[**{**b, a = Definition2[x], c = "Options[" <> ToString[x] <> "]"**}**, b = a[[–1]]; ClearAllAttributes[x]; ClearAll[x]; ToExpression[Select[a, StringFreeQ[ToString[#], c] &]]; If[**{**y**}**==** {}**, If[b !=** {}**, SetAttributes[x, b]]]]**

In[2581] **:= DeleteOptsAttr[G]**
In[2582]**:= Definition2[G]**
Out[2582]= {**"**G[x**_**, y_]**:=** x^2+ y^2**"**, {Listable**,** Protected}}
In[2583]**:= Vs[x_, y_] := x^2 + y^2; Options[Vs] =** {**v–> 72, g–> 67**}**;** In[2584]**:= SetOptions[Vs,** {**v–> 72, g–> 67**}**]; SetAttributes[Vs, Protected]** In[2585]**:= Definition2[Vs]**
Out[2585]= {**"**Vs[x**_**, y_]**:=** x^2+ y^2**"**, **"**Options[Vs]**:=** {v–> 71**,** g–> 66}**"**,

{Protected}}
In[2586]**:= DeleteOptsAttr[Vs, 590]**
In[2587]**:= Definition2[Vs]**
Out[2587]= {**"**Vsv[x**_**, y_]**:=** x+ y**"**, {}}

At that, it must be kept in mind that the given procedure isn 't applicable to the standard system functions, returning on them the unevaluated call as it well illustrates the following example, namely**:**

In[2618] **:=** {**DeleteOptsAttr[Sin, 90], DeleteOptsAttr[Sin]**}
Out[2618]= {DeleteOptsAttr[Sin**,** 78]**,** DeleteOptsAttr[Sin]}
For certain built–in**Mathematica** functions, for example**Plot,** are ascribed the options whose values can be redetermined. At that, if at a function call the values for its admissible options aren't defined, then for them values by default are used. The function call**Options[*F,op*]** allows to obtain values by default for an option***op*** of a function***F,*** for example**:**

In[2620] **:= Options[Plot,** {**PlotLabel, FrameStyle, PlotStyle, PlotRange, ColorOutput**}**]**
Out[2620]= {PlotLabel–> None**,** FrameStyle–> {}**,** PlotStyle–> Automatic**,** PlotRange–> {Full**,** Automatic}**,** ColorOutput–> Automatic}

The mechanism of options which is supported by the **Mathematica** can be successfully used in development of both the applications of various type, and a separate software. The interested reader can familiarize oneself with this mechanism more in details in rather well–developed help–base of the **Mathematica** system, or in books [31-33,52,61,59].

In conclusion of the present section we in brief will stop on application of transformations rules to the procedures. The mechanism of transformations rules supported by the system remains in force not only for symbols but for algebraic expressions too. In principle, this mechanism can be adapted onto an arbitrary expression. Moreover, as an essential enough property of this mechanism it is possible to note the circumstance that allows to use and the patterns, and the symbolic constructions, for example**:**

In[2837]**:= Sin[x^2]^2 + Cos[y + h]^2 /.**{**x^2–> x, y + h–> x**}

Out[2837] = Cos[x]^2+ Sin[x]^2

In[2838]:= **Sin[a + b*c]*(Sin[x^2] + Cos[y + h]) /. {Sin[_]–> x, Cos[_]–> y}**
Out[2838]= x (x+ y)
Thus, between purely symbolic transformations rules and rules that include the patterns, in particular,**"_"** there is one fundamental difference which is considered in details in the books [31-33]. The more detailed description of mechanisms of programming of the patterns for transformations rules of an arbitrary expression can be found in the reference on the**Mathematica**. At the same time the system doesn't dispose the mechanism of application of transformations rules to procedures and for this purpose the procedure can be offered, whose call**ReplaceProc[*x, t*]** returns the definition in string format of the procedure– result of application to a procedure***x*** of rules of transformations***t****(one rule or their list)****;*** at that, those rules are excluded from rules***t,*** whose left parts coincide with formal arguments of the procedure***x.*** The next fragment represents source code of the procedure with examples of its use along with the simple testing function whose call**RuleQ[*x*]** returns ***True,*** if***x*** – a transformation rule, and***False*** otherwise. In the**ReplaceProc** procedure definition the given function is used in its heading.

In[2859] **:= RuleQ[x_] := If[MemberQ[{Rule, RuleDelayed}, Head[x]], True, False]**
In[2860]:= **Map[RuleQ, {a–> b, c–> d+h, Sin, a+b, ProcQ, a :> b, c :> d+h}]**
Out[2860]= {True, True, False, False, False, True, True}

In[2861] **:= ReplaceProc[x_ /; ProcQ[x],**
r_ /; DeleteDuplicates[Map[RuleQ, Flatten[{r}]]] == {True}] := Module[{a = Definition2[x], b = HeadPF[x], c, d = Flatten[{r}]},

c = ToExpression["Hold[" <> StringTrim[a[[1]], b <> " := "] <> "]"]; d = Select[d, ! MemberQ[Args1[x], ToString[Part[#, 1]]] &]; c = ToString1[ReplaceAll[c, d]]; b <> " := " <> StringTake[c, {6,–2}]]

In[2862] **:= ArtKr[x_ /; IntegerQ[x], y_ /; StringQ[y]] :=**
Module[{a = StringLength[y], b = 90, ab = 500}, (a + x)*(b + y) + ab]
In[2863]:= **ReplaceProc[ArtKr, {a–> Art, b–> Kr, y–> 42, x–> 500}]**
Out[2863]= "ArtKr[x_ /; IntegerQ[x], y_ /; StringQ[y]]:= Module[{Art= StringLength[y], Kr= 90, ab= 500}, (Art+ 500)*(Kr+ 42)+ ab]" It makes sense to stop on one moment useful for programming. Supra, a number of procedures which return additional result through argument– *an undefinite variable*– were considered. However such mechanism requires or of choice of some undefinite variable in the current session, demanding generally of its cleaning in the absence of need for it, or saving of value of a certain variable with subsequent its cleaning in a procedure and restoration of an initial value before any exit from the procedure. Meanwhile, for this purpose the similar mechanism which is based on the**UniqueV** procedure can be used, whose call**UniqueV[*x,y*]** returns a name in string format***"xn"*** of an unique variable of the current session to which value***y*** was ascribed, where***x*** – a symbol,***n*** – an integer and***y*** – an arbitrary expression. Further the**UniqueV** procedure is used for ensuring of return of additional result by simple procedure***A6*** through an unique variable. The fragment is rather transparent and of any special additional explanations doesn't demand.

In[2571]:= **UniqueV[x_ /; SymbolQ[x], y_] :=**
Module[{a = ToString[Unique[ToString[x]]]}, ToExpression[a <> " = " <> ToString1[y]]; a]

In[2572] **:= UniqueV[“agn”, 50090]**
Out[2572]= “agn20”
In[2573]**:= agn20**
Out[2573]= 50 090
In[2579]**:= A6[x_, y___] := Module[**{**a = 90, b = 500**},

If[{y}== {}, **a*x,** {**a*x, UniqueV[“ag”, b*x]**}**]]** In[2580]**:=** {**A6[73], A6[42, 6]**}
Out[2580]= {6570, {3780, “ag68”}}
In[2581]**:= ag68**
Out[2581]= 21 000

Below, a number of tools providing higher level of *procedural* programming in the**Mathematica** system will be represented**;** which in a certain degree were wafted by similar means of the**Maple** system and by other systems of procedural programming. It should be noted that the values by default for system attributes and options given in the present section, and in the book as a whole concern the**Mathematica** system of version***10.*** In the following **Mathematica** versions there can quite be certain differences.

6.9. Some additional facilities for operating with blocks, functions and modules in the*Mathematica*software

If the previous sections of the head represent main means of work with an object of the type {***Block, Function, Module***}, the present section represents additional, but quite important means of work in a number of appendices with objects of this type. Meanwhile, having the basic purpose, these means can be functionally crossed with means represented in the previous sections of this head. It should not cause any particular surprise because the similar situation takes place pretty often among means, practically, of any software system. And still the means of the present section have a little more specific character and aren’t so sought after as means of the previous sections. At the same time, ascribing them to this section in a certain degree has conditional character and is caused by our experience of their usage.

First of all, again we will return to the question of syntactic correctness of a block and module. Examples of two types of syntactic mistakes at definition of the procedures of types {***Module, Block***} are presented below, that aren’t distinguished by the system at evaluation of their definitions, and in some cases even at a call of such procedures. At that, repeated calls of procedures of the***Module***type as very much demonstrate the fragment examples, yield formally correct results. For testing of procedures of both types regarding their*syntactic* correctness in the above context the**SyntCorProcQ** procedure has been offered, whose source code along with typical examples of its use the following fragment represents, namely**:**

In[5081] **:= Art[x_, y_] := Module[**{**a, b**}**,]; Art1[x_, y_] := Module[**{**a, b**}**]** In[5082]**:= Kr[x_, y_] := Block[**{**a, b**}**,]; Kr1[x_, y_] := Block[**{**a, b**}**]** In[5083]**:=** {**Art[90, 500], Art1[90, 500]**}
Module**::**argr**:** Module called with 1 argument;2 arguments are expected**.** >> Out[5083]= {Null, Module{a, b}]}
In[5084]**:=** {**Art[90, 500], Art1[90, 500]**}

Out[5084]= {Null, Module[{a, b}]}
In[5085]:= {**Kr[90, 500], Kr1[90, 500]**}

Block::argr: Block called with 1 argument; 2 arguments are expected. >> Out[5085]= {Null, Block[{a, b}]}
In[5086]:= **SyntCorProcQ[x_ /; BlockModQ[x]] := Module[{d, h, c = Kr, b = PureDefinition[x], a = HeadPF[x]},**

ClearAll[Kr]; Kr = ProcFuncTypeQ[ToString[x]][[2]][[1]]; h = Quiet[Check[Locals2[x], Locals1[x]]]; h = If[h === {}, "{}", ToString[h]]; d = a <> " := " <> Kr <> "[" <> h; d = StringReplace[b, d–> "", 1]; Kr = c; ! MemberQ[{"]", ", Null]"}, d]]

In[5087] := **Map[SyntCorProcQ, {ProcQ, Kr, Kr1, Art, Art1}]**
Out[5087]= {True, False, False, False, False}
In[5088]:= **KrArt[x_, y_, z_] := Module[{a, b, c}, 90 + x + y + z]** In[5089]:= **Map[SyntCorProcQ, {Locals, Mapp, BlockToModule, KrArt}]** Out[5089]= {True, True, True, True}
In[5090]:= **Map[SyntCorProcQ, {Art2, Do, If, 500}]**
Out[5090]= {SyntCorProcQ[Art2], SyntCorProcQ[Do], SyntCorProcQ[If],

SyntCorProcQ[500]}

The procedure call **SyntCorProcQ[*x*]** returns*True* if the definition of a block or module***x*** activated in the current session is syntactic correct in the above context, otherwise*False* is returned. If***x*** **–** not a block or module, the call is returned unevaluated. The definition of the**SyntCorProcQ** procedure along with the standard means uses our means such as**ProcFuncTypeQ, Locals2, BlockModQ,PureDefinition** and**HeadPF** that are considered in the present book and in [28-33]. In a number of problems of procedural programming, first of all, of system character, the procedure is useful enough. In a number of applications of system character it is desirable for the user block, function or module to have information regarding use by it of means in the context {*system means,user means*}. The**SysUserSoft** procedure solves this problem, whose the call**SysUserSoft[*x*]** generally returns the nested***2*****–** element list, whose first element contains***2*****–**element sublists, whose the first element**–** the name in string format of a system function, and the second element**–** its multiplicity, while the second element of the list also contains ***2*****–**element sublists, whose first element**–** the name in string format of the user means*(block,function,module)***,** and the second element**–** its multiplicity. In the absence for an object***x*** means of the specified types the procedure call **SysUserSoft[*x*]** returns the empty list, i.e. {}. At that, if the type of the actual argument***x*** is different from***(Block, Function, Module),*** then the procedure call**SysUserSoft[*x*]** is returned unevaluated. The next fragment represents source code of the procedure along with typical examples of its usage.

In[2580] := **SysUserSoft[x_ /; BlockFuncModQ[x]] := Module[{b, s ={}, u ={}, h = Args[x, 6], c, a = Flatten[{PureDefinition[x]}][[1]], d = If[QFunction[x], {}, LocalsGlobals1[x]]}, b = ExtrVarsOfStr[a, 2, 90]; c = Select[b, ! MemberQ[Flatten[{ToString[x], h, "True", "False", "$Failed", Quiet[d[[1]]], Quiet[d[[3]]]}], #] &]; Map[If[Quiet[SystemQ[#]], AppendTo[s, #], If[BlockFuncModQ[#], AppendTo[u, #]]] &, c]; c = Map[Gather, {s, u}]; c =**

{Map[Flatten[#] &, Map[{#, Length[#]}&, c[[1]]]], Map[Flatten[#] &, Map[{#, Length[#]}&, c[[2]]]]}; c = {Map[DeleteDuplicates[#] &, c[[1]]], Map[DeleteDuplicates[#] &, c[[2]]]}; If[Flatten[c] == {}, {}, c]]

In[2581] := **A[m_, n_, p_ /; IntegerQ[p], h_ /; PrimeQ[h]] := Module[{a = 73}, h*(m + n + p)/a + StringLength[ToString1[z]]/(Cos[c] + Sin[d])]**
In[2582]:= **SysUserSoft[A]**
Out[2582]= {{{"Cos", 1}, {"IntegerQ", 1}, {"Module", 1}, {"PrimeQ", 1}, {"Sin", 1}, {"StringLength", 1}}, {{"ToString1", 1}}}
In[2583]:= **SysUserSoft[SysUserSoft]**
Out[2583]= {{{"AppendTo", 2}, {"DeleteDuplicates", 2}, {"Flatten", 5}, {"Gather", 1}, {"If", 4}, {"Length", 2}, {"MemberQ", 1}, {"Module", 1}, {"Quiet", 3}, {"Select", 1}, {"ToString", 1}}, {{"Args", 1}, {"BlockFuncModQ", 2}, {"ExtrVarsOfStr", 1}, {"LocalsGlobals1", 1}, {"PureDefinition", 1}, {"QFunction", 1}, {"SystemQ", 1}}}
In[2584]:= **G[x_] := x^2 + 90*x + 500; SysUserSoft[G]**
Out[2584]= {}
In[2585]:= **F[x_] := a*x + Sin[b*x] + StringLength[ToString1[x + c]]; SysUserSoft[F]**

Out[2585] = {{{"Sin", 1}, {"StringLength", 1}}, {{"ToString1", 1}}} In[2586]:= **SysUserSoft[QFunction]**
Out[2586]= {{{"Block", 1}, {"CompiledFunction", 1}, {"If", 5}, {"MemberQ", 1},

{ "Module",2},{"Quiet",2},{"StringJoin",1},{"StringReplace",2}}, {{"Definition2", 1}, {"HeadPF", 2}, {"Map3", 1}, {"SingleDefQ", 1},
{"SuffPref", 4}, {"ToString1", 1}, {"ToString3", 1}}}

As showed our expirience, the**SysUserSoft** procedure is rather useful in the structural analysis of the user software of types {***Block, Function, Module***}.

In some cases the **RenBlockFuncMod** procedure is a rather interesting tool of manipulation by the blocks, functions or modules of the same name. The procedure call**RenBlockFuncMod[*x, y*]** returns a new name of a function/ block/module***x*** in string format determined by the format**Unique[*y*]<>*H*,** where***y*** **–** a symbol, whereas***H*** **–** one of symbols {***"B", "F", "M"***} depending on type of an object***x*** or of type of its subobject composing it in case of the object***x*** of the same name. At that, an object***x*** is removed from the current session whereas the result of such renaming keeps options and attributes of the source object***x*****.** The fragment represents source code of the procedure along with typical examples of its usage.

In[2526]:= **Pr[x_, y_String, z_ /; If[z === 90, True, False]] := {x, y, z}; Pr[x_, y_ /; StringQ[y], z_ /; If[z === 90, True, False]] :=**

Module[{}, {x, y, z}]; SetAttributes[Pr, Protected]; Pr1[x_, y_String, z_ /; If[z === 90, True, False]] := {x, y, z}; SetAttributes[Pr1, {Protected, Listable}]

In[2527]:= **RenBlockFuncMod[x_ /; BlockFuncModQ[x], y_Symbol] :=**

Module[{t = {}, h, a = Options[x], b = Attributes[x], k = 1, n, c = Flatten[{PureDefinition[x]}], d = Flatten[{HeadPF[x]}]}, For[k, k <= Length[c], k++, h = StringReplace[c[[k]], StringJoin[d[[k]], " := "]–> ""]; h = If[SuffPref[h, "Module[{", 1], "M", If[SuffPref[h, "Block[{", 1], "B", "F"]]; n =

ToString[Unique[y]] <> h; AppendTo[t, n]; ToExpression[StringReplace[c[[k]], ToString[x] <> "[“–> n <> "[", 1]]; If[a != {}, ToExpression["SetOptions[" <> n <> ", " <> ToString[a] <> "]"]]; If[b !={}, ToExpression["SetAttributes[" <> n <> ", " <> ToString[b] <>"]"]]]; ClearAllAttributes[x]; ClearAll[x]; If[Length[t] == 1, t[[1]], t]]

In[2528] **:= RenBlockFuncMod[Pr1, Sv**]
Out[2528]= "Sv$66130F"
In[2529]**:= Definition["Sv$66130F"]**
Out[2529]= Attributes[Sv$66130F]= {Listable, Protected}

Sv $66130F[x_, y_String, z_ /; If[z=== 90, True, False]]:= {x,y,z} In[2530]:= **RenBlockFuncMod[Pr, Sv]**
Out[2530]= {"Sv$66731F", "Sv$66733M"}
In[2531]**:= Definition["Sv$66731F"]**
Out[2531]= Attributes[Sv$66731F]= {Protected}

Sv $66731F[x_, y_String, z_ /; If[z=== 90, True, False]]:= {x,y,z} In[2532]:= **Definition[Sv$66733M]**
Out[2532]= Attributes[Sv$66733M]= {Protected}

Sv $66733M[x_, y_String, z_ /; If[z=== 90, True, False]]:= Module[{}, {x, y, z}]
In[2533]**:= Map[Definition, {Pr, Pr1}]**
Out[2533]= {Null, Null}

The **RenBlockFuncMod** procedure is most of all convenient in case of need of differentiating of an object***x*** of the same name onto the single subobjects composing it. In certain cases at the procedures calls which are in the user's package*(files of the types* {***"cdf",“m", "mx"***}) that is uploaded into the current session, their *local* variables, including local variables of the*nested* procedures, in the field of the**Mathematica** variables are associated with the context ascribed to the given package. This mechanism the more in details here isn't considered. It also concerns the symbolical results returned by a procedure of this package through such local variables. In this case the symbolical result accepts the following standard format, namely:

<Context ascribed to a package>`<Procedure name>`Result

For the purpose of*elimination* of the similar situation and receiving so-called *reduced result (that contains no forms**a`b`**)* that is significantly better adapted for the subsequent processing, to a result returned by a procedure of the user package, can be applied the**ReductRes** function whose call**ReductRes[*x, a*]** returns the reduced result***a*** returned by a procedure***x*** of the user package that has been loaded into the current session. The next fragment represents both variants of the**Head1** procedure without usage and with usage of such mechanism with an illustration of results of the call of both procedures. The received results rather visually illustrate a basic distinction arising from the mechanism of reduction of results on the basis of the presented**ReductRes** function. The following fragment represents source code of the function.

In[3282]**:= ReductRes[x_ /; SymbolQ[x], y_] := ToExpression[StringReplace[ToString[y], Context[x] <> ToString[x] <> "`"–> ""]]**

In[3283] **:= ReductRes[Head1, AladjevProcedures`Head1`System]** Out[3283]= System

In[3284]:= **Map[Head, {ProcQ, Sin, 90, a+b, Function[{x, y}, x+y], G[x], J[6], Head1 }]**
Out[3284]= {Symbol, Symbol, Integer, Plus, Function, G, J, Symbol} In[3285]:= **Map[Head1, Map[ToString, {ProcQ, Sin, 90, a + b,**

Function[{x, y}, x + y], G[x], J[6], Head1}]] Out[3285]= {"Module", "System", "Integer", "Plus", "PureFunction", "G", "J",

" Module"}
In[3286]:= **Head1[a := b]**
Out[3286]= AladjevProcedures`Head1`System

The following useful **Avg** procedure is internal, i.e. the procedure call**Avg[]** makes sense only in the body of other procedure, returning a list of nesting {***1***|***2***} whose elements define the***2***–element lists whose first elements define local variables in string format of a procedure, external in relation to the**Avg** whereas the second– their initial values in string format; at that, lack of the initial value is coded by the symbol***"None".*** In case of more than one local variable the***ListList***–list is returned, whose sublists have the above format. At absence for external procedure of local variables the procedure call**Avg[]** returns the empty list– {}. The call**Avg[]** outside of other procedure doesn't make special sense, returning the list of the above format for***2****local variables* {***a, b***} of the**Avg** procedure as visually illustrates the following fragment.

In[2723]:= **Avg[] := Module[{b, a = ToString[ToExpression[ToString[InputForm[Stack[_][[1]]]]]]}, a = If[! SuffPref[a, {"Module[", "Block["}, 1], "Module[{}," <> a <> "]", a];**

a = StringReplace[a, "$" –> ""]; a = StringReplace[a, If[SuffPref[a, "Block[", 1], "Block[", "Module["]–> "", 1]; a = SubStrSymbolParity1[a, "{", "}"][[1]]; If[a == "{}", {}, b = StrToList[StringTake[a, {2,–2}]]; b = Map[StringSplit[#, " = "] &, b]; Map[If[Length[#] == 1, {#[[1]], "None"}, #] &, b]]]

In[2724] **:= Z[m_, n_, p_ /; IntegerQ[p]] := Module[{h, x = 90, y = {a, b}}, m+ n + p; h = Avg[]; h]**
In[2725]**:= Z[73, 90, 500]**
Out[2725]= {{"h", "None"}, {"x", "90"}, {"y", "{a, b}"}}
In[2726]**:= G[m_, n_, p_ /; IntegerQ[p]] := Module[{a, b = 73, c, d = 90}, d = Avg[]; m + n + p; d]**
In[2727]**:= G[t, p, 500]**
Out[2727]= {{"a", "None"}, {"b", "73"}, {"c", "None"}, {"d", "90"}}
In[2728]**:= A[m_, n_, p_ /; IntegerQ[p], h_ /; PrimeQ[h]] := Module[{a = 500.90, b, c, t, q, d = 73, z = 47}, b = Avg[]; m + n + p + h; m*n; b]**
In[2729]**:= A[x, y, 42, 47]**
Out[2729]= {{"a", "460.78"}, {"b", "None"}, {"c", "None"}, {"t", "None"}, {"q", "None"}, {"d", "73"}, {"z", "47"}}
In[2730]**:= B[m_, n_, p_, h_ /; PrimeQ[h]] := Module[{a = 500.90, b, c = {h, p}, t, q, d = 73, z = p*t, s}, b = Avg[]; m + n + p + h; m*n; b]**
In[2731]**:= B[x, y, 42, 47]**
Out[2731]= {{"a", "500.90"}, {"b", "None"}, {"c", "{47, 42}"}, {"t", "None"}, {"q",

"None"}, {"d", "73"}, {"z", "42 t"}, {"s", "None"}}
In[2732]:= **T[m_, n_, p_, h_ /; PrimeQ[h]] := Module[{}, m*n*p*h; Avg[]]; T[25, 18, 42, 47]**
Out[2732]= {}
In[2733]:= **Avg[]**
Out[2733]= {{"b", "None"},
{"a", "ToString[ToExpression[ToString[Stack[_][[1]]]]]"}}

The previous fragment represents source code of the **Avg** procedure with examples of its usage for receiving in the body of a procedure of the list of its local variables. It should be noted that a number of system means of our package***AVZ_Package*** [48] use the**Avg** procedure.

Here once again quite pertinently to note the important circumstance, that the blocks, functions, modules differ by their headings as it was repeatedly illustrated above. At that, at the call of an object of this type the first of the complete list of the subobjects of the same name determined by the standard **Definition** function is choosen on which the tuple of the actual arguments is admissible. This circumstance should be considered at programming and it has been considered by us at programming of a number of means of our package***AVZ_Package*** [48]. Moreover, as objects of the same name can be as objects of type {***Block, Function, Module***}, and in combination with objects of other types, in a number of cases at calculations with such objects, causing special or erroneous situations. For elimination from the objects of the same name of subobjects of types different from {***Block,Function, Module***} a quite simple procedure serves whose call**ProcCalls[*w*]** returns*Null*, i.e. nothing, deleting from the list of the object of the same name*w(win string format)* of subobjects of types, different from {***Block, Function, Module***}. The following fragment represents source code of the**ProcCalls** procedure along with the most typical examples of its usage.

In[2780] := **A[x_] := Module[{a = 50}, x + a]; A[x_, y_] := Module[{a = 90}, x + y + a]; A[x_, y_List] := Block[{}, {x, y}]; A[x_Integer] := Module[{a = 42}, x + a]; A := {a, b, c, d, h}; SetAttributes[A, {Flat, Listable, Protected}]** In[2781]:= **Definition[A]**

Out[2781] = Attributes[A]= {Flat, Listable, Protected}
A:= {a, b, c, d, h}
A[x_Integer]:= Module[{a= 42}, x+ a]
A[x_]:= Module[{a= 50}, x+ a]
A[x_, y_List]:= Block[{}, {x, y}]
A[x_, y_]:= Module[{a= 90}, x+ y+ a]

In[2782]:= **ProcCalls[x_/; StringQ[x]] :=
Module[{a = Select[StringSplit[ToString[InputForm[Definition[x]]], "\n"],**

# != " " && #!= x && ! SuffPref[#, x <> " := ", 1] &]}, If[SuffPref[a[[1]], "Attributes[", 1], AppendTo[a[[2 ;;–1]], a[[1]]]]; ClearAttributes[x, Protected]; Clear[x]; Map[ToExpression, a];]

In[2783] := **ProcCalls["A"]**
In[2784]:= **Definition[A]**
Out[2784]= Attributes[A]= {Flat, Listable, Protected}

A[x _Integer]**:=** Module[{a= 42}**,** x+ a]
A[x_]**:=** Module[{a= 50}, x+ a]
A[x_**,** y_List]**:=** Block[{}, {x, y}]
A[x_**,** y_]**:=** Module[{a= 90}, x+ y+ a]

In[2785]**:= ScanLikeProcs[x_: {}] := Module[{b = {}, c = {}, d, h, k = 1, a = Select[Names["`*"], StringFreeQ[#, "$"] &&**

Quiet[Check[BlockFuncModQ[#], False]] &]}, Off[Definition::ssle]; If[a == {}, Return[{}], For[k, k <= Length[a], k++, d = Definition2[a[[k]]][[1 ;;–2]]; If[Length[d] > 1, AppendTo[b, Map[StringTake[#, {1, Flatten[StringPosition[#, " := "]][[1]]–1}] &, d]];

AppendTo[c, a[[k]]]]]]; On[Definition::ssle]; If[! HowAct[x], x = b, Null]; c]

In[2786] **:= G[x_] := Module[{a = 500}, x^2 + a]; G[x_ /; PrimeQ[x]] := Module[{a = 90}, x + a]; G[x_, y_] := Module[{}, x + y]; G[x_, y_ /; ListQ[y], z_] := Module[{}, x + Length[y] + z]**

In[2787] **:= V[x_] := Module[{}, x]; V[x_ /; ListQ[x]] := Module[{}, Length[x]]**
In[2788]:= {**ScanLikeProcs[], ScanLikeProcs[Sv], Sv**}
Out[2788]= {{"A"**,** "G"**,** "V"}, {"A"**,** "G"**,** "V"}, {{"A[x_Integer]"**,** "A[x_**,** y_List]"**,**

"A[x_**,** y_]"**,** "A[x_]"}, {"G[x_ /; PrimeQ[x]]"**,** "G[x_]"**,** "G[x_**,** y_]"**,** "G[x_**,** y_ /; ListQ[y], z_]"}, {"V[x_ /; ListQ[x]]"**,** "V[x_]"}}}

In addition to the previous procedure for the purpose of determination of the blocks**,** functions**,** modules of the same name of the current session of the system a quite simple procedure is intended whose the call**ScanLikeProcs[]** returns the list of the blocks/functions/modules of the same name that are activated in the current session while as a result of the call**ScanLikeProcs[*b*]** in addition thru an undefinite variable***b*** the list of headings in string format of objects of the specified type is returned. The previous fragment represents source code of the**ScanLikeProcs** procedure along with examples of its use. In certain appendices these means are rather useful, above all, at elaboration of the system means for manipulations with procedures.

In a number of cases the*structural* analysis of objects of type {***Block, Module, Function***} represents the undoubted interest. In connection with this the next **StructProcFunc** procedure providing a certain structural analysis of objects of this type was created. The fragment below represents the**StructProcFunc** procedure whose call**StructProcFunc[*x*]** returns simple or nested list whose elements depending on type {***"Block", "Module", "Function"***} of an actual argument***x*** have format {*Type*,*Heading*,*Locals*,*Body*} for {***"Block", "Module"***} and {*Type*,*Heading*,*Body*} for***"Function";*** furthermore, qua of the function is understood an object***x*** such as**BlockFuncModQ[*x*] =*True*.** This fragment represents source code of the procedure along with examples of its usage off which the format of the result returned by the procedure is highly obvious.

In[3223] **:= StructProcFunc[x_ /; BlockFuncModQ[x]] := Module[{c, d, h={}, p, k = 1, t, b = Flatten[{HeadPF[x]}], a = Flatten[{PureDefinition[x]}]}, c = Map9[StringReplace, a, Map[StringJoin[#, " := "]–> "" &, b]]; While[k <= Length[b], d = c[[k]]; If[SuffPref[d, "Module[{", 1], t = "Module",**

If[SuffPref[d, "Block[{", 1], t = "Block", t = ""]]; If[t != "", AppendTo[h, {t, b[[k]], p = SubStrSymbolParity1[d, "{", "}"][[1]];

StrToList[p], StringReplace[StringTake[d, {1,–2}], t <> "[" <> p <> ", "–> ""]}], AppendTo[h, {"Function", b[[k]], StringReplace[d, b[[k]] <> " := "–> ""]}]]; k++]; If[Length[h] == 1, h[[1]], h]]

In[3224] **:= Agn[x_] := Block[{a = 90, b =500}, x^2*a*b]; Agn[x_, y_] := x+y**
In[3225]**:= Agn[x_, y_, z_] := Module[{a = 90}, a*(x + y + z)]**
In[3226]**:= StructProcFunc[Agn]**
Out[3226]= {{"Block", "Agn[x_]", {"a= 90", "b= 500"}, "x^2*a*b"},

{ "Function", "Agn[x_, y_]", "x+ y"},
{"Module", "Agn[x_, y_, z_]", {"a= 90"}, "a*(x+ y+ z)"}} In[3227]**:= Avz[x__] := Module[{a =6, b = Stack[_]}, a+x; b; $InBlockMod]** In[3228]**:= StructProcFunc[Avz]**
Out[3228]= {"Module", "Avz[x__]", {"a = 6", "b= Stack[_]"},
"a+ x; b; $InBlockMod"}
For the purpose of elimination of ambiguity of the modules, functions and blocks of the same name it is recommended to apply standard means to the *cleaning* of the current session off concrete definitions, using the cancellation of the***Protected***attribute for them if it is necessary. For cleaning of symbols off the ascribed values the**Mathematica** has three functions**Clear, ClearAll** and**Remove** that are considered, for example, in [32]. However, the given functions demand the concrete designation of the symbols that are subject to the cleaning off the ascribed expressions. Whereas the following fragment represents source code of the**ClearCS** procedure with examples of its usage whose call**ClearCS[*ClearAll*]** returns*Null***,** i.e. nothing, clearing all symbols and off the ascribed values received by them in the current session, and off attributes, messages and values by default, associated with such symbols; while the call**ClearCS[*Remove*]** returns*Null***,** i.e. nothing, deleting from the field of names of the system all symbols that received values in the current session of the**Mathematica** system.

In[2640]**:= ClearCS[x_ /; MemberQ[{ClearAll, Remove}, x]] := Module[{a = Join[Names["Global`*"], {"a", "b", "c", "d", "h", "k", "p", "S", "x", "y"}]}, Quiet[Mapp[ClearAttributes, a, Protected]]; Quiet[Map[x, a]];]** In[2641]**:= {x, y, z, g, h}= {42, 73, 47, 68, 2015}; ClearCS[Remove]; {x,y,z,g,h}** Out[2641]= {Removed[x], Removed[y], Removed[z], Removed[g], Removed[h]}
In[2642]**:= {x, y, z, g, h}= {42, 73, 47, 68, 2015}; ClearCS[ClearAll];**

{ x, y, z, g, h}
Out[2642]= {x, y, z, g, h}
In[2643]**:= G[x_] := Module[{a = 90}, x^2 + a]; V[x_] := Module[{}, x^2];**

G[x_ /; PrimeQ[x]] := Module[{a = 500}, x + a];
V[x_ /; ListQ[x]] := Module[{}, Length[x]]
In[2644]**:= ClearCS[ClearAll]; Map[Definition, {G, V}]**
Out[2644]= {Null, Null}

In certain appendices the **ClearCS** procedure appears as an useful enough means, in many respects providing recovery of initial status of the current session. The procedure is used by some means of package ***AVZ_Package*,** carrying out the function of preliminary cleaning of the current session. In the problems of formal processing of functional expressions the**ExpArgs** procedure represents a quite certain interest whose call**ExpArgs[*G,*{*x,y,* …}]** provides extension of the list of formal arguments of a module, function or block***G*** onto the list of arguments {***x, y, z,* …**} to the right concerning a tuple of formal arguments of the object***G*** with return of*Null* value, i.e. nothing, and with activation in the current session of the updated definition of the object***G.*** The expansion of a tuple of formal arguments is made for object***G*** only onto variables from the list {***x, y,* …**} which aren't its formal arguments or local variables; otherwise expansion isn't made. List elements {***x, y, z,* …**} onto updating can be symbols in string format along with names of formal arguments with tests for*admissibility* of the corresponding actual arguments ascribed to them. At that, the procedure call**ExpArgs[*G, x*]** on inadmissible object***G,*** in particular, on a system function or on the*empty* list***x*** is returned unevaluated. The following fragment represents source code of the**ExpArgs** procedure along with some most typical examples of its usage.

In[2547] **:= A[x_] := Module[{a = 6}, x*a]; A[x_, y_] := Module[{a = 7}, x*y*a]; A[x_, y_List] := Block[{}, {x, y}]; A[x_Integer] := Module[{a = 5}, x*a]; SetAttributes[A, {Flat, Listable, Protected}]; Art[x_, y_ /; PrimeQ[y]] := Module[{a = 2, b = 6}, Length[Join[x, y]]*a*b]**

In[2548]:= **ExpArgs[f_ /; BlockFuncModQ[f], x_ /; ListQ[x] && DeleteDuplicates[Map[! StringFreeQ[ToString[#], "_"] ||**

StringQ[#] &, x]] == {True}] := Module[{a, b, c, d, t, h, g={}, k = 1}, a = Flatten[{Definition4[ToString[f]]}]; b = Args[f, 90]; b = If[NestListQ[b], b[[1]], b]; d = Locals1[f]; d = If[NestListQ[d], d[[1]], d];

c = Flatten[{HeadPF[f]}][[1]]; t = Map[ToString, x]; h = Map[#[[1]] &, Map[StringSplit[#, "_"] &, t]]; b = Join[b, d]; While[k <= Length[h], If[! MemberQ[b, h[[k]]], d = t[[k]]; AppendTo[g, If[StringFreeQ[d, "_"], d <> "_", d]]]; k++]; If[g == {}, Return[], g = ToString[g]; d = StringTake[c, {1,–2}] <> ", " <> StringTake[g, {2,–2}] <> "]"; ClearAllAttributes[f]; ClearAll[f]; a[[1]] = StringReplace[a[[1]], c–> d, 1]; Map[ToExpression, a]];] In[2549]:= **ExpArgs[Art, {"x", "z_", "h", p_ /; String[p], c_String, h_ /; ListQ[h] && Length[h] >= 90}]**
In[2550]:= **Definition[Art]**

Out[2550] = Art[x_, y_ /; PrimeQ[y], z_, h_, p_ /; String[p],h_ /; ListQ[h]**&&** Length[h]>= 90]:= Module[{a= 2, b= 6}, Length[Join[x, y]] a b]
In[2551]:= **ExpArgs[Art, {"x", "z_", "h", p_ /; String[p], c_Integer, h_ /; ListQ[h] && Length[h] >= 90}]**
In[2552]:= **Definition[Art]**
Out[2552]= Art[x_, y_ /; PrimeQ[y], z_, h_, p_ /; String[p], c_String, h_ /; ListQ[h]**&&** Length[h]>= 90]:= Module[{a= 2, b= 6}, Length[Join[x, y]] a b]
In[2553]:= **ExpArgs[A, {"x", "z_", "h", p_ /; String[p], c_Integer, h_ /; ListQ[h] && Length[h] >= 90}]**
In[2554]:= **Definition[A]**
Out[2554]= Attributes[A]= {Flat, Listable, Protected}
A[x_Integer, z_, h_, p_ /; String[p], c_Integer, h_ /; ListQ[h]**&&** Length[h]>= 90]:= Module[{a= 5}, x a] A[x_]:= Module[{a= 6}, x a]

A[x_, y_List]:= Block[{}, {x, y}]
A[x_, y_]:= Module[{a= 7}, x y a]
In[2555]:= **ExpArgs[A, {"x", "z_", "h", p_ /; String[p], c_Integer, h_ /; ListQ[h] && Length[h] >= 90}]**
In[2556]:= **Definition[A]**
Out[2556]= Attributes[A]= {Flat, Listable, Protected}
A[x_Integer, z_, h_, p_ /; String[p], c_Integer, h_ /; ListQ[h] && Length[h]>= 90]:= Module[{a= 5}, x a] A[x_]:= Module[{a= 6}, x a]
A[x_, y_List]:= Block[{}, {x, y}]
A[x_, y_]:= Module[{a= 7}, x y a]

Definition of the **ExpArgs** procedure along with the standard means uses a series of our means such as**Args, BlockModQ, ClearAllAttributes, HeadPF, Definition4, Locals1, NestListQ** that are considered in the present book and in [33]. The**ExpArgs** procedure has a series of rather interesting appendices, first of all, applications of the system character.

The next fragment represents the useful procedural variable **$ProcType** that has been implemented by a simple function on the basis of the system**Stack** function and making sense only in the body of a block or module, returning type {***Block, Module***} in string format of an object containing it. Outside of objects of the specified type the variable accepts the"ToString" value which doesn't have especial meaning. The next fragment represents source code of the**$ProcType** variable along with some typical examples of its usage. The **$ProcType** variable has a number of rather useful appendices of the applied and the system character.

In[2562]:= **$ProcType := ToString[Stack[][[1]]]**

In[2563] := **Agn[x_, y_] := Block[{a = 90, b = 500, c = $ProcType}, a + b + c; {$ProcType, c}]**
In[2564]:= **Agn[42, 47]**
Out[2564]= {"Block", "Block"}
In[2565]:= **Agn[x_, y_] := Module[{a = 90, b = 500, c = $ProcType}, a + b + c; {$ProcType, c}]**
In[2566]:= **Agn[42, 47]**
Out[2566]= {"Module", "Module"}
In[2567]:= **Agn[x_, y_] := Module[{c = $ProcType, a = 90, b = 500}, a + b + c; {$ProcType, c}]**
In[2568]:= **Agn[42, 47]**
Out[2568]= {"Module", "Module"}
In[2569]:= **$ProcType**
Out[2569]= "ToString"

To the previous procedural variable another procedural variable **$TypeProc** directly adjoins which is also used only in the body of a block or module of any type. The variable**$TypeProc** receives value of type in string format of an object***G*** which contains it, in the context {***"Block", "DynamicModule", "Module"***}; outside of a block or module the variable receives**$Failed** value as clearly illustrates the fragment representing source code of the procedural variable**$TypeProc** along with examples of its most typical usage.

In[2572] **:= $TypeProc := CheckAbort[If[$a25k18$ =Select[{Stack[Module], Stack[Block], Stack[DynamicModule]}, # != {}&]; If[$a25k18$ == {}, Clear[$Art24$Kr17$]; Abort[], $a25k18$ = ToString[$a25k18$[[1]][[1]]]]; SuffPref[$a25k18$, "Block[{", 1], Clear[$a25k18$]; "Block", If[SuffPref[$a25k18$, "Module[{", 1] && ! StringFreeQ[$a25k18$, "DynamicModule"], Clear[$a25k18$]; "DynamicModule", Clear[$a25k18$]; "Module"]], $Failed]**

In[2573] **:= M[x_] := Module[{a = 90, b = 500, c = $TypeProc}, c]; M[73]** Out[2573]= "Module"
In[2574]:= **G[x_] := Module[{a = 6, b =7, c}, c =a*b*x; c^2; $TypeProc]; G[73]**
Out[2574]= "Module"
In[2575]**:= B[x_] := Block[{a = 90, b = 500, c = $TypeProc}, c]; B[68]** Out[2575]= "Block"
In[2576]**:= DM[x_] := DynamicModule[{a, c = $TypeProc}, x; c]; DM[68]** Out[2576]= "DynamicModule"
In[2577]**:= $TypeProc**
Out[2577]= $Failed
In[2578]**:= F[x_ /; ListQ[x]] := Append[Select[x, OddQ[#] &], $TypeProc];**

F[{68, 73, 47, 18}]
Out[2578]= {73, 47, $Failed}
In certain cases of procedural programming the**$TypeProc** variable along with the**$ProcType** variable are useful enough facilities.

To the previous procedural variables the **$CallProc** variable directly adjoins whose call returns*contents* in string format of the body of a block or module which contains it at the time of a call. At that, for a module the body with local variables with**"$"** symbols ascribed to them while for a block its body in the standard format are returned. The call of the given variable outside of a block or module returns**"**StringTake[ToString1[Stack[_][[1]]]**,**{10, 2}]**".**The next fragment represents source code of the procedural variable**$CallProc** along with the typical examples of its usage.

In[2584]**:= $CallProc := StringTake[ToString1[Stack[_][[1]]], {10,–2}]**

In[2585] **:= M[x_, y_ /; StringQ[y]] := Module[{a = $CallProc, b, c}, x*StringLength[y]; a]**
In[2586]**:= M[6, "vak"]**
Out[2586]= **"**Module[{a$ = $CallProc, b$**,** c$}, 6*StringLength["vak"]**;** a$]**"**
In[2587]**:= B[x_, y_ /; PrimeQ[y]] := Block[{a = $CallProc, b}, x + y; a]** In[2588]**:= B[500, 17]**
Out[2588]= **"**Block[{a= $CallProc, b}**,** 500+ 17; a]**"**
In[2589]**:= $CallProc**
Out[2589]= **"**StringTake[ToString1[Stack[_][[1]]], {10**,–**2}]**"**

The procedural variable **$CallProc** provides possibility of processing of the body of a block or a module, containing it, within the confines of the given object, presenting a certain interest for a number of applications, first of all, of the system character.

Use of means of preservation of definitions in the *ASCII* format files allows to program quite effective and useful means of the analysis of the structural organization of the user blocks, functions and modules. The next fragment represents source code of the**CompActPF** procedure along with the typical examples of its application, whose call**CompActPF[*x*]** returns the nested***2*–** element list whose the first element defines the list of all blocks**,**functions or modules that enter in the definition of a block/function/module***x*,** including ***x*** whereas the second element defines the list of headings in string format of these means. At that, the lists include only the user means whose definitions were activated in the current session of the**Mathematica** system**;** moreover, for the calls which enter into an object***x*,**are added respectively and all their calls onto the full depth of nesting.

In[5134]**:= G[x_] := Module[{}, a*x + b]; G1[x_] := a*x + b + V[x, 90]; S[y_] := Module[{}, y^2 + 90]; S1[y_] := y^2 + G[y]; V[x_, y_] := Module[{G, S}, G[x] + S[y^2]]; V1[x_, y_] := G1[x] + S1[y^2] + h*Sin[x*y] + v*Cos[x*y]**

In[5135] **:= CompActPF[x_ /; BlockFuncModQ[x]] := Module[{b ={}, c ="", d, a = ToDefOptPF[x], f = ToString[x] <> ".txt", h = ""}, Put[FullDefinition[x], f]; Quiet[While[! SameQ[h, EndOfFile], h = Read[f, String]; If[h != " ", c = c <> h; If[HeadingQ[d =StringTake[c,{1, Flatten[StringPosition[c, " := "]][[1]]–1}]], AppendTo[b, d]; c = ""]; Continue[]]]]; DeleteFile[Close[f]]; {Map[HeadName, b], b}]**

In[5136]**:= CompActPF[V1]**

Out[5136] = {{"V1", "G1", "V", "G", "S", "S1"}, {"V1[x_, y_]", "G1[x_]", "V[x_, y_]", "G[x_]", "S[y_]", "S1[y_]"}}
In[5137]**:= CompActPF[V]**
Out[5137]= {{"V", "G", "S"}, {"V[x_, y_]", "G[x_]", "S[y_]"}}

In[5138] **:= CompActPF1[x_ /; BlockFuncModQ[x]] := Module[{d = {}, k = 1, b = Args[x, 90], a = Flatten[{PureDefinition[x]}][[1]], c = Locals1[x], p}, {b, c}= {If[NestListQ[b], b[[1]], b], If[NestListQ[c], c[[1]], c]}; a = Select[ExtrVarsOfStr[a, 2], ! MemberQ[Flatten[{ToString[x], Join[b, c, {"Block", "Module"}]}], #] &]; While[k <= Length[a], p = a[[k]]; AppendTo[d, If[BlockFuncModQ[p], {p, HeadPF[p]}, If[SystemQ[p], {p, "System"}, {p, "Undefined"}]]]; k++]; a = Map[Flatten, Gather[d, ! StringFreeQ[#1[[2]], "_"] && ! StringFreeQ[#2[[2]], "_"] &]]; b = Map[Flatten, Gather[a, #1[[2]] =="System" && #2[[2]]== "System" &]]; d = Map[Flatten, Gather[b, #1[[2]] == "Undefined" && #2[[2]] == "Undefined" &]]; Map[If[#[[–1]] == "System",**

Prepend[MinusList[#, {"System"}], "System"], If[#[[–1]] == "Undefined", Prepend[MinusList[#, {"Undefined"}], "Undefined"], #]] &, d]] In[5139]**:= CompActPF1[V1]**
Out[5139]= {{"System", "Cos", "Sin"}, {"G1", "G1[x_]", "S1", "S1[y_]"},

{ "Undefined", "h", "v"}}
In[5140]**:= CompActPF1[V]**
Out[5140]= {}
In[5141]**:= Z[x_/; StringQ[x], z_ /; ! HowAct[x]] := Block[{a = Sin[x]},**

Cos[a] + StringLength[x]] In[5142]**:= CompActPF1[Z]**
Out[5142]= {{"System", "Cos", "Sin", "StringLength"},

{”HowAct“, “HowAct[x_]”}}

We will note, that for effective processing of the saved complete definitions of functions, blocks and modules in definition of the**CompActPF** procedure the procedure has been used, whose the call**ToDefOptPF[*x*]** optimizes the definition of the user block, function or module***x*** in the current session. The truth, for optimization of definitions there are also other means considered in the book above. A quite useful modification of the**CompActPF** procedure completes the previous fragment whose the call**CompActPF1[*x*]** returns the nested list whose elements represent sublists of the following format**:**

– sublist with the first element***“System”*** defines calls of system functions in definition of a block, function or module***x;***
– sublist with the first element***“Undefined”*** defines names of objects which aren**’**t included into the list of arguments and local variables of a function, block or module***x;***
– sublist of a format different from above-mentioned contains the user pairs {block/function/module**,** its heading}**,** whose calls are available in definition of an object***x.***

The **CompActPF1** procedure is an useful means in a number of applications, first of all, of the system character, providing the structural analysis of the user means of the types {***Block, Function, Module***}.

As it is well known [25], the **Maple** system has a number of the procedural variables*(where under procedural variables are understood the variables making sense only in the body of a block or module and receiving values about components of the object containing them)* which provide, in particular, the possibility to receive the list of formal arguments of the block or module in its body at a call. Whereas in the**Mathematica** system similar means are absent though in many cases represent quite certain interest. Some means of this kind for Mathematica are given above. It is simple to notice that means of the**Maple** in this respect are more developed, than similar means of the**Mathematica,** that in some cases rather significantly simplifies procedural programming.

A block or module provide four main mechanisms of return of results of its call**:*(1)*** through the*last* offer of the body**, *(2)*** on the basis of**Return** function, ***(3)*** through global variables, and***(4)*** through formal arguments. The given question was considered enough in detail in our books [25-33]. The following fragment on the example of rather simple procedure***P*** very visually illustrates a mechanism of return of any number of results through argument**z –** the tuple of undefinite variables. At that, for simplification of assignment of the returned results to elements of a list**z** a simple and at the same time useful function**AssignL** is used.

```
In[2550]:= P[x_, y_, z___ /; DeleteDuplicates[Map[! HowAct[#] &, {z}]] == {True}]
:= Module[{a = 90, b = 500, c = 72},

If[x*y > 500, AssignL[ {z}[[1]], a]; AssignL[{z}[[2]], b]; AssignL[{z}[[3]], c]]; (x +
y)*(a + b + c)] In[2551]:= P[42, 47, m, n, p]
Out[2551]= 58 918
In[2552]:= {m, n, p}
Out[2552]= {90, 500, 72}
In[2553]:= First[{x, y, z}] = 90
```

Set ::write: Tag First in First[{x, y, z}] is Protected. >> Out[2553]= 90
In[2554]:= **{x, y, z}**
Out[2554]= {x, y, z}
In[2555]:= **{x, y, z}[[2]] = 90**

Set ::setps: {x, y, z} in the part assignment is not a symbol. >> Out[2555]= 90
In[2556]:= **{x, y, z}**
Out[2556]= {x, y, z}

In[2557]:= **AssignL[x_, y_, z___] := Quiet[If[{z}!= {}, x := y, x = y]]**

In[2558] := **AssignL[{x, y, z}[[2]], 90]**
Out[2558]= 90
In[2559]:= **{x, y, z}**
Out[2559]= {x, 90, z}
In[2560]:= **AssignL[{a1, a2, a3, a4, a5, a6}[[3 ;; 5]], {72, 47, 67}]**
Out[2560]= {72, 47, 67}
In[2561]:= **{a1, a2, a3, a4, a5, a6}**
Out[2561]= {a1, a2, 72, 47, 67, a6}
In[2562]:= **AssignL[{{a, b}, {c, d}}[[1, 2]], 90, Delayed]**
In[2563]:= **{{a, b}, {c, d}}**
Out[2563]= {{a, 90}, {c, d}}
In[2564]:= **AssignL[{{a, b}, {c, d}}[[1, 2]], 90, Delayed]**
Ot[2564]= $Failed
The function call**AssignL[*x, y*]** provides correct assignment to elements*(to all or the given elements)* of an arbitrary expression or expressions from the list***y*,** modeling assignments on the basis of constructions of the format {***x, y, z,…***}**[[*n*]]= *Expr*** and {***x, y, z,…***}**[[*n* ;; *p*]]=** {***Ex***$_n$,***Ex***$_{n+1}$, …,***Ex***$_p$} and to them similar which the system doesn't support while the function call**AssignL[*x, y, j*]** where***j*** – an expression– provides the correct delayed assignments of the above-stated kind, as visually illustrates the previous fragment. At that, the function call on inadmissible appointments returns**$Failed.** As it was already noted earlier and it was used in some procedures, in the **Mathematica** along with the simple procedures which aren't containing in the body of definitions of other procedures the use of the so–called***nested*** procedures, i.e. of such procedures whose definitions are in body of other procedures is allowed. The nesting level of such procedures is defined by only a size of working field of the system. In this regard rather interesting problem of definition of the list of subprocedures whose definitions are in the body of an arbitrary procedure of the type {***Block, Module***} arises. The **SubProcs** procedure successfully solves the problem whose call**SubProcs[*x*]** returns the nested***2***-element list of***ListList***-type whose*first* element defines the sublist of headings of blocks and modules composing a main procedure ***x*** whereas the*second* element defines the sublist of the generated names of blocks and modules composing a main procedure***x*** including procedure***x*** itself, and that are activated in the current session of**Mathematica** system. The following fragment represents source code of the**SubProcs** procedure along with the most typical examples of its application.

In[2525] := **SubProcs[P_ /; BlockModQ[P]] := Module[{b, c = {}, d, t, h, k = 1, p = {}, g = {}, a = Flatten[{PureDefinition[P]}][[1]]}, b = StringPosition[a, {"] := Block[{", "] := Module[{"}]; For[k, k <= Length[b], k++, d = b[[k]]; AppendTo[p,**

ExprOfStr[a, d[[1]],–1, {" ", ",", ";"}]]; AppendTo[c, h = ExprOfStr[a, d[[1]],–1, {" ", ",", ";"}] <> " := " <> ExprOfStr[a, d[[1]] + 5, 1, {" ", ",", ";"}]; t = Flatten[StringPosition[h, "["]]; h = Quiet[StringReplacePart[h, ToString[Unique[ToExpression[StringTake[h, {1, t[[1]]–1}]]]], {1, t[[1]]–1}]]; AppendTo[g, StringTake[h, {1, Flatten[StringPosition[h, "["]][[1]]–1}]]; h]]; Map[ToExpression, c]; {p, Map[ToExpression, g]}]

In[2526]:= **P[x_, y_] := Module[{a, b, B, P1, P2}, P1[z_, h_] := Module[{m, n}, z+h]; B[h_] := Block[{}, h]; P2[z_] := Module[{P3},**

P3[h_] := Module[{}, h]; P3[z]]; x*P2[x] + P1[x, y] + P2[y]] In[2527]:= **P[90, 500]**
Out[2527]= 9190
In[2528]:= **SubProcs[P]**
Out[2528]= {{"P[x_, y_]", "P1[z_, h_]", "B[h_]", "P2[z_]", "P3[h_]"},

{P $60501, P1$60506, B$60510, P2$60515, P3$60519}} In[2529]:= **DefFunc[P2$60515]**
Out[2529]= P2$1247[z_]:= Module[{P3}, P3[h_]:= Module[{}, h]; P3[z]]

Thus, between elements of sublists of the returned nested list the one-to-one correspondence takes place. The definition of the**SubProcs** procedure along with the standard means uses a number of our means such as**BlockModQ, ExprOfStr, PureDefinition** which are considered in the present book and in [28–33]. The procedure allows a number of interesting enough expansions.

The quite useful **SubProc1** procedure provides testing of a block/module*x* regarding existence in its definition of the blocks/modules. The procedure call**SubProcs1[*x*]** depending on existence of an object*x* of the same name with various headings or with one heading returns the nested or simple list; at that, the*first* elements of the list or sublists define headings of an object*x* while the*second* define number of blocks/modules that enter into definition of the object*x* with the corresponding headings. If the object*x* not a block, function, module, the procedure call**SubProcs1[*x*]** is returned unevaluated. The fragment represents source code of the procedure with the most typical examples of its use. The**SubProcs1** procedure can be quite simply expanded onto extraction of all subprocedures of a procedure***x*.**

In[2532] **:= SubProcs1[x_ /; BlockFuncModQ[x]] := Module[{b ={}, c, d, k = 1, a = Flatten[{PureDefinition[x]}]}, For[k, k <= Length[a], k++, c = a[[k]]; d = StringPosition[c, {"] := Module[{", "] := Block[{"}]; If[d == {}, Continue[]]; AppendTo[b, {StringTake[c, {1, d[[1]][[1]]}], Length[d]–1}]]; If[Length[b] == 1, Flatten[b], b]]**

In[2533] **:= G[x_, y_, z_] := x + y + z; G[x_] := Module[{V, H}, V[y_] := Module[{}, y^3]; H[z_] := Module[{}, z^4]; x + V[x] +H[x]]; G[x_, z_] := Module[{V, H, P}, V[t_] := Module[{}, t^3 + t^2 + 500]; H[t_] := Module[{}, t^4]; P[h_] := Module[{a = 90}, a^2 + h^2];**

x + V[x] + H[z]*P[x]];

H[t_] := Module[{P}, P[h_] := Module[{a = 90}, a^2 + h^2]; x + P[x]] In[2534]:= **SetAttributes[G, {Protected, Listable}]; {G[2015], G[2015, 73]}** Out[2534]= {16 493

608 406 015, 115 541 459 232 440}
In[2535]:= **SubProcs1[G]**
Out[2535]= {{"G[x_]", 2}, {"G[x_, z_]", 3}}
In[2536]:= **SubProcs1[H]**
Out[2536]= {"H[t_]", 1}
In[2537]:= **SubProcs1[90]**
Out[2537]= SubProcs1[90]
In[2538]:= **P[x_ /; {j[b_] := Module[{}, b^2], If[EvenQ[x], True, False]}[[2]]] :=**

Module[{a = {c[d_] := Module[{}, d]}}, {j[x], c[x]}] In[2539]:= **P[2014]**
Out[2539]= {4056196, 2014}
In[2540]:= **Map[Definition1, {j, c}]**
Out[2540]= {"j[b_]:= Module[{}, b^2]", "c[d_]:= Module[{}, d]"}

Very simple example illustrating some admissible mechanisms of definition of heading and local variables of a block/module that are enough useful for procedural programming completes this fragment. These mechanisms are used also by a number of the means composing our package***AVZ_Package*** [48] while the**SubProcs2** procedure represents a quite essential expansion of the**SubProcs1** procedure. The following fragment represents source code of the**SubProcs2** procedure along with examples of its typical usage.

In[2386] := **G[x_] := Module[{V, H}, Vg[y_] := Module[{}, y^3]; H72[z_] := Module[{}, z^4]; x + Vg[x] + H72[x]]; G[x_, z_] := Module[{Vt, H, P}, Vt[t_] := Module[{}, t^3 + t^2 + 500]; H[t_] := Module[{}, t^4]; P[h_] := Module[{a = 90}, a^2 + h^2]; x + Vt[x] + H[z]*P[x]];**

H[t_, z_] := Module[{P}, P[h_] := Module[{a = 90}, a^2 +h*z]; t +P[t]]; F[x_, y] := x + y; SetAttributes[G, {Protected, Listable}]; {G[2015], G[2015, 73]} Out[2386]= {16 493 608 406 015, 115 541 459 232 440}
In[2387]:= **SubProcs2[y_, z___] := Module[{n = {}, m=1, SB,**
v = Flatten[{PureDefinition[y]}]},

If[BlockFuncModQ[y], SB[x_String] := Module[{b = "Module[", c, d, h, g = "", t, k, p, q, j, s, w, a = Map[#[[1]] &, StringPosition[x, "Module[{"]]}, If[a == {}, Return[]]; If[Length[a] == 1, Return[$Failed], d = Map[#–5 &, a]]; c = {StringTake[x, {1, d[[1]]}]}; For[k = Length[a], k > 1, k—, h = b; g = ""; t = ""; For[j = a[[k]] + 7, j < Infinity, j++, h = h <> StringTake[x, {j, j}]; If[SameQ[Quiet[Check[ToExpression[h], "Error"]], "Error"], Continue[], For[j = d[[k]], j > 1, j—, g = StringTake[x, {j, j}] <> g; If[SameQ[Quiet[Check[ToExpression[g], "Error"]], "Error"], Continue[], Break[]]]; While[j > 1, p = StringTake[x, {j, j}]; If[! SameQ[p, " "], t = p <> t, Break[]]; j—]; p = StringPosition[x, " " <> t <> "["][[1]]; s = Flatten[SubStrSymbolParity1[StringTake[x, {p[[1]],–1}], "[", "]"]]; w = 1; While[w <= Length[s]–1, q = s[[w]]; If[! StringFreeQ[q, "_"], s = t <> q <> " := Module" <> s[[w + 1]]; Break[]]; w++]; AppendTo[c, s]; Break[]]]]; c]; For[m, m <= Length[v], m++, AppendTo[n, SB[v[[m]]]]]; n =Select[n, ! SameQ[#, Null] &]; If[n =={}, $Failed, n =If[Length[n]== 1, n[[1]], n]; If[{z}!= {}, ToExpression[n]]; n], $Failed]]

In[2388]:= **SubProcs2[G, 90]**
Out[2388]= {{"G[x_]", "H72[z_]:= Module[{}, z^4]", "Vg[y_]:= Module[{}, y^3]"},

{"G[x_, z_]", "P[h_]:= Module[{a= 90}, a^2+ h^2]", "H[t_]:= Module[{}, t^4]", "Vt[t_]:= Module[{}, t^3+ t^2+ 500]"}}

In[2389] := {**H72[90], Vg[500], P[26], H[18], Vt[67]**}
Out[2389]= {65 610 000, 125 000 000, 8 776, 104 976, 305 752}
In[2390]:= **SubProcs2[H]**
Out[2390]= {{"H[t_, z_]", "P[h_]:= Module[{a= 90}, a^2+ h*z]"}, $Failed} In[2391]:= **Map[SubProcs2, {F, 500}]**
Out[2391]= {$Failed, $Failed}

The call **SubProcs2[*y*]** depending on an unique procedure*y* or of the same name with various headings, returns simple or nested list. For the returned list or sublists the first element is the procedure*y* heading, while the others
– definitions in string format of subprocedures of the***Module*** type that enter into the*y* definition. In absence for*y* of subprocedures of the specified type or in the case of type of argument*y*, different from***Module***, the procedure call**SubProcs2[*y*]** returns**$Failed.** In case of the second optional argument ***z*** –*an arbitrary expression*– the call**SubProcs3[*y, z*]** returns the similar result with simultaneous activation of these subprocedures in the current session. The**SubProcs3** procedure is further expansion of the**SubProcs2** procedure; its call**SubProcs3[*y*]** differs from a call**SubProcs2[*y*]** by the following two moments, namely:***(1)*** the user block, function or module can act as a factual argument***y***, and***(2)*** the returned list as the first element contains heading of object***y*** whereas other elements of the list represent definitions of functions, blocks and modules in string format entering into definition***y.*** In case of an object***y*** of the same name, the returned list will be the nested list, sublists of which have the above–mentioned format. At that, the call**SubProcs3[*y, z*]** with the second optional argument***z*** – an arbitrary expression– returns the above list and at the same time activates in the current session all objects of the above type, that enter into***y.*** The fragment represents source code of the procedure along with typical examples of its usage.

In[2630]:= **G[x_] := Module[{Vg, H72},Vg[y_] := Module[{},y^3]; H72[z_] := Module[{}, z^4]; x + Vg[x] + H72[x]]; G[x_, z_] := Module[{Vt, H, P},**

Vt[t_] := Module[{}, t^3 + t^2 + 500]; H[t_] := Module[{}, t^4]; P[h_] := Module[{a = 90}, a^2 +Cos[h^2]]; Sin[x] + Vt[x]+ H[z]*P[x]]; H[t_] := Module[{P}, P[h_] := Module[{a = 90}, a^2*h^2]; Cos[t]*P[t]]; F[x_, y_] := Sin[x + y] + Cos[x–y]; V[x_] := Block[{a, b, c}, a[m_] := m^2; b[n_] := n + Sin[n]; c[p_] := Module[{}, p]; a[x]*b[x]*c[x]]; SetAttributes[G, {Protected, Listable}]

In[2631]:= **SubProcs3[y_, z___] := Module[{u = {}, m = 1, Sv, v = Flatten[{PureDefinition[y]}]}, If[BlockFuncModQ[y],**

Sv[S_String] := Module[{a = ExtrVarsOfStr[S, 1], b, c = {}, d, t = 2, k = 1, cc = {}, n, p, j, h = {StringTake[S, {1, Flatten[StringPosition[S, " := "]][[1]]–1}]}}, a = Select[a, ! SystemQ[Symbol[#]] && ! MemberQ[{ToString[G]}, #] &]; b = StringPosition[S, Map[" " <> # <> "[" &, a]]; p = Select[a, ! StringFreeQ[S, " " <> # <> "["] &]; b = Flatten[Map[SubStrSymbolParity1[StringTake[S, {#[[1]],–1}], "[", "]"] &, b]]; For[j = 1, j <= Length[p], j++, n = p[[j]]; For[k = 1, k <= Length[b]–1, k++, d = b[[k]]; If[! StringFreeQ[d, "_"] && StringTake[b[[k + 1]], {1, 1}] == "[", AppendTo[c, Map[n

<> d <> " := " <> # <> b[[k+1]] &, {"Block", "Module"}]]]]]; c = DeleteDuplicates[Flatten[c]]; For[k = 1, k <= Length[c], k++, d = c[[k]]; If[! StringFreeQ[S, d], AppendTo[h, d], AppendTo[cc, StringTake[d, {1, Flatten[StringPosition[d, " := "]][[1]]–1}]]]]]; {h, cc}= Map[DeleteDuplicates, {h, cc}]; p = Map[StringTake[#, {1, Flatten[StringPosition[#, "["]][[1]]}] &, h]; cc = Select[Select[cc, ! SuffPref[#, p, 1] &], ! StringFreeQ[S, #] &]; If[cc == {}, h, For[k = 1, k <= Length[cc], k++, p = cc[[k]]; p = StringCases[S, p <> " := " ~~ __ ~~ "; "]; AppendTo[h, StringTake[p, {1, Flatten[StringPosition[p, ";"]][[1]]–1}]]]]]; Flatten[h]]; For[m, m <= Length[v], m++, AppendTo[u, Sv[v[[m]]]]]; u = Select[u, ! SameQ[#, Null] &]; u = If[Length[u] == 1, u[[1]], u]; If[{z}!= {}, ToExpression[u]]; u, $Failed]] In[2632]:= **SubProcs3[G]**
Out[2632]= {{"G[x_]", "Vg[y_]:= Module[{}, y^3]", "H72[z_]:= Module[{},

z ^4]"}, {"G[x_, z_]", "Vt[t_]:= Module[{}, t^3+ t^2+ 500]", "H[t_]:= Module[{}, t^4]", "P[h_]:= Module[{a= 90}, a^2+ Cos[h^2]]"}} In[2633]:= **SubProcs3[H]**
Out[2633]= {"H[t_]", "P[h_]:= Module[{a= 90}, a^2*h^2]"}
In[2634]:= **SubProcs3[F]**
Out[2634]= {"F[x_, y_]"}
In[2635]:= **SubProcs3[V]**
Out[2635]= {{"V[x_ /; ListQ[x]]"}, {"V[x_]", "c[p_]:= Module[{}, p]",

" a[m_]:= m^2", "b[n_]:= n+ Sin[n]"}, {"V[x_, y_]"}} In[2636]:= **SubProcs3[V, 500]**
Out[2636]= {{"V[x_ /; ListQ[x]]"}, {"V[x_]", "c[p_]:= Module[{}, p]",

" a[m_]:= m^2", "b[n_]:= n+ Sin[n]"}, {"V[x_, y_]"}} In[2637]:= {**V[90], a[42], b[47], c[67]**}
Out[2637]= {729000 (90+ Sin[90]), 1764, 47+ Sin[47], 67}

If a function with heading acts as an object ***y***, only its heading is returned; the similar result takes place and in case of an object***y*** that doesn't contain subobjects of the above type whereas on an object***y*** different from the user block, function or module, the call of the**SubProcs3** returns**$Failed.**

In some cases there is a necessity of definition for a block and module of the subobjects of the type {***Block,Function, Module***}. The call**SubsProcQ[*x, y*]** returns*True* if***y*** is a global active subobject of an object***x*** of the above type, and*False* otherwise. But as the***Math–***objects of the given type differ not by names as that is accepted in the majority of programming systems, but by headings then through the***3rd*** optional argument the procedure call returns the nested list whose sublists as*first* element contain headings with a name ***x*** while the*second* element contain the headings of subobjects corresponding to them with a name***y.*** On the first***2*** arguments {***x,y***} of the types, different from specified in a procedure heading, the procedure call**SubsProcQ[*x, y*]** returns*False.* The next fragment represents source code of the**SubsProcQ.**

In[2650]:= **SubsProcQ[x_, y_, z___] := Module[{a, b, k = 1, j = 1, Res = {}}, If[BlockModQ[x] && BlockFuncModQ[y], {a, b}= Map[Flatten, {{Definition4[ToString[x]]}, {Definition4[ToString[y]]}}];**

For[k, k <= Length[b], k++, For[j, j <= Length[a], j++, If[! StringFreeQ[a[[j]], b[[k]]], AppendTo[Res, {StringTake[a[[j]], {1, Flatten[StringPosition[a[[j]], " := "]]

[[1]]–1}], StringTake[b[[k]], {1, Flatten[StringPosition[b[[k]], " := "]][[1]]–1}]}], Continue[]]]]; If[Res != {}, If[{z}!= {}&& ! HowAct[z], z = If[Length[Res] == 1, Res[[1]], Res]; True], False], False]]

In[2651]:= **V[x_] := Block[{a, b, c}, a[m_] := m^2; b[n_] := n + Sin[n]; c[p_] := Module[{}, p]; a[x]*b[x]*c[x]]; c[p_] := Module[{}, p];**

V[x_, y_] := Module[{a, b, c}, a[m_] := m^2; b[n_] := n + Sin[n]; c[p_] := Module[{}, p]; a[x]*b[x]*c[x]]; c[p_] := Module[{}, p]; p[x_] := x; SetAttributes[V, Protected]

In[2652] **:=** {**SubsProcQ[V, c, g67], g67**}
Out[2652]= {True**,** {{"V[x_]"**,** "c[p_]"}, {"V[x**_,** y_]"**,** "c[p_]"}}}
In[2653]**:= SubsProcQ[V, Avz]**
Out[2653]= False
In[2654]**:= SubsProcQ[Sin, h]**
Out[2654]= False
In[2655]**:= SubsProcQ[p, c]**
Out[2655]= False

In principle, on the basis of the above five means { **SubProcs ÷ SubProcs3, SubsProcQ**} it is possible to program a number of useful enough means of operating with expressions of the types {***Block, Module***}.

In a certain regard the procedural variable **$ProcName** which is used only in the body of a procedure activated in the current session is of interest**;** the variable returns the list whose first element determines a name whereas the second element**–** the heading in string format of the procedure containing it. Moreover, for providing of the given possibility in a list of local variables of a procedure containing**$ProcName** variable it is necessary to encode the expression of the type**$$NameProc$$ =*"Procedure_Name",*** otherwise the procedure call as a value of variable**$ProcName** returns***"UndefinedName".*** The following fragment represents source code of the procedural variable **$ProcName** along with typical examples of its usage.

In[2530]**:= $ProcName := Module[**{**d = "$$ArtKr$$", a, b, c, t = "", k**}**, a = ToString1[Stack[_]]; d = Flatten[StringPosition[a, d]][[1]]; b = Flatten[StringPosition[a, "$$NameProc$$"]][[1]];**

If[b > d || ToString[b] == "", Return["UndefinedName"], k = b]; For[k = b, k <= d, k++, c = StringTake[a, {k, k}]; If[MemberQ[{"," , "}"}, c], Break[], t = t <> c; Continue[]]]; {b = ToExpression[ToExpression[StringSplit[t, "="][[2]]]], HeadPF[b]}]

In[2531] **:= Avz[x_, y_, z_] := Module[**{**$$NameProc$$ = "Avz", b**}**, b = $ProcName; x+y+z; b]**
In[2532]**:= Agn[x_, y_, z_] := Module[**{**b, $$NameProc$$ = "Agn"**}**, x+y+z; b = $ProcName; b]**
In[2533]**:= Ian[x_, y_, z_] := Module[**{**b, c, h**}**, x+y+z; b = $ProcName; b]**
In[2534]**:= Agn[47, 67, 72]**
Out[2534]= {Agn**,** "Agn[x**_,** y**_,** z_]"}
In[2535]**:= Avz[47, 67, 72]**

Out[2535]= {Avz, "Avz[x_, y_, z_]"}
In[2536]:= **Ian[47, 67, 72]**
Out[2536]= "UndefinedName"

This variable in a certain degree was wafted by the procedural ***"procname"*** variable of the**Maple** system which plays quite essential part, first of all, in procedural programming of various problems of the system character.

The **BFMSubsQ** procedure represents a quite certain interest; the procedure call**BFMSubsQ[*x*]** returns the list of format {***True, Heading***} if definition of the user block or module***x*** contains definitions of blocks, functions and/or modules, otherwise the list {***False, Heading***} is returned. In case of an object of the same name***x*** of the above type the call returns the nested list whose sublists have the specified format. On an object***x*** of a type, different from {***Block, Module***}, the procedure call returns*False.* At that, the procedure call **BFMSubsQ[*x, y*]** with the*2nd* optional argument***y*** *–an undefinite variable–* through***y*** returns the list of format {***Heading, N***} where***N*** defines number of blocks, functions and modules that enter into a subobject with the heading ***Heading*** of an object of the same name***x.*** The following fragment represents source code of the**BFMSubsQ** procedure along with a number of the most typical examples of its usage.

In[2545]:= **G[x_] := Module[{Vg,H7}, Vg[y_] := Module[{}, y^3]; H7[z_] := Module[{}, z^4]; x+Vg[x] + H7[x]];**

G[x_, z_] := Module[{Vt, H, P}, Vt[t_] := Module[{}, t^3 + t^2]; H[t_] := Module[{}, t^4]; P[h_] := Module[{a = 6}, a^2 + Cos[h^2]]; Sin[x]+Vt[x]+H[z]*P[x]]; H[t_] := Module[{P}, P[h_] := Module[{a =6}, a^2 + h]]; T[x_] := Block[{a}, a[y_] := y^2; x + a[500]]; T[x_, y_] := Module[{a = 6}, x*y + a* Cos[t] + P[t]]; F[x_, y_] := Sin[x/y] + Cos[x*y]; SetAttributes[G, {Protected, Listable}]

In[2546] := **BFMSubsQ[x_, y___] := Module[{a, b, c, d = {}, k =1, p, h, g ={}}, If[! BlockModQ[x], False,**
{a, b}= Map[Flatten, {{PureDefinition[x]}, {HeadPF[x]}}];

For[k, k <= Length[a], k++, p = a[[k]]; p = StringReplace[p, b[[k]] <> " := "–> "", 1]; c = Select[ExtrVarsOfStr[p, 1], ! SystemQ[#] &]; h = Flatten[Map[StrSymbParity[p, " " <> #, "[", "]"] &, c]]; h = Select[h, SuffPref[#, Map[StringJoin[" " <># <> "["] &, c], 1] && ! StringFreeQ[#, "_"] &]; AppendTo[g, {b[[k]], Length[h]}]; AppendTo[d, {If[h != {}, True, False], b[[k]]}]]; If[{y}!= {}&& ! HowAct[y], y = g]; If[Length[d] == 1, d[[1]], d]]]

In[2547] := **BFMSubsQ[H]**
Out[2547]= {True, "H[t_]"}
In[2548]:= **BFMSubsQ[G]**
Out[2548]= {{True, "G[x_]"}, {True, "G[x_, z_]"}}
In[2549]:= **BFMSubsQ[T]**
Out[2549]= {{True, "T[x_]"}, {False, "T[x_, y_]"}}
In[2550]:= **Map[BFMSubsQ, {F, 90, Agn, Sin}]**
Out[2550]= {False, False, False, False}
In[2551]:= **BFMSubsQ[G, g]**

Out[2551]= {{True, "G[x_]"}, {True, "G[x_, z_]"}}
In[2552]:= **g**
Out[2552]= {{"G[x_]", 2}, {"G[x_, z_]", 3}}
The definition of the**BFMSubsQ** procedure along with the standard means uses a number of our means such as**BlockModQ, PureDefinition, HeadPF, HowAct, ExtrVarsOfStr, StrSymbParity, SuffPref** and**SystemQ** which are considered in the present book and in [30,33]. The procedure generalizes and expands the above procedures**SubProcsQ ÷ SubProcsQ3** and**SubsProcQ;** the**BFMSubsQ** procedure is useful enough in a number of the appendices connected with processing of procedures of type {***Module, Block***} and, first of all, of the system character.

On the basis of the procedures**BlockModQ, HeadPF, Mapp, PureDefinition** and**SubStrSymbolParity1** that are considered in the present book, also the useful**ProcBody** procedure has been programmed whose call**ProcBody[*x*]** returns the body in string format of the user block, module and function***x*** with heading. The procedure successfully processes also the objects of the same name***x*,** returning the list of bodies of subobjects composing object***x*.** The following fragment represents source code of the**ProcBody** procedure along with typical examples of its usage.

In[2093] **:= ProcBody[x_ /; BlockFuncModQ [x]] := Module[{c, p, d ={}, k =1, a = Flatten[{PureDefinition[x]}], b = Flatten[{HeadPF[x]}]}, While[k <= Length[a], p = a[[k]]; c =Mapp[Rule, Map[b[[k]]<>" :=" <> #&, {"Block[", "Module[", ""}], ""]; c = StringReplace[p, c, 1]; AppendTo[d, If[BlockModQ[x], StringTake[StringReplace[c, SubStrSymbolParity1[c, "{", "}"][[1]] <> ", "–> "", 1], {1,–2}], c]]; k++]; If[Length[d] == 1, d[[1]], d]]**

In[2094] **:= Art[x_, y_, z_] := Module[{a=x+y+z, c ={m, n}, b = 90}, a^2+a+b]**
In[2095]:= **ProcBody[Art]**
Out[2095]= "a^2+ a+ b"
In[2096]:= **T[x_] := Block[{a}, a[y_] := y^2; x + a[90]];**

T[x_, y_] := Module[{a = 500}, x*y + a]
In[2097]:= **ProcBody[T]**
Out[2097]= {"a[y_]:= y^2; x+ a[90]", "x*y+ a"}
In[2098]:= **F[x_, y_] := x + y + x*y; F[x] := Sin[x] + x*Cos[x]; ProcBody[F]**
Out[2098]= {"Sin[x]+ x*Cos[x]", "x+ y+ x*y"}
The**ProcBody** procedure plays a rather essential part in a number of tasks of the procedural programming dealing with various manipulations with definitions of functions and procedures of type {***Block, Module***} along with components composing them.
In a number of the tasks caused by a processing of string representation of definitions of the user procedures and blocks the questions of partition of this representation onto***2*** main components– the procedure body and its frame with the final procedural bracket**"]"** a certain interest can represent. In this context and the**PartProc** procedure can be quite useful. Procedure call**PartProc[*x*]** returns the two–element list, whose first element in string format represents a procedure frame with the final procedural bracket**"]";** the place of the body of a procedure is taken by the substring***"Procedure Body"*** whereas the second element of the list in string format represents a procedure body***x*.** Furthermore, as a procedure frame the construction of the format**"Heading := Module[{*locals*}, …]"** is

understood. In the case of erroneous situations the procedure call is returned unevaluated or returns **$*Failed.***The next fragment represents source code of the**PartProc** procedure along with typical examples of its usage.

In[2049]**:= PartProc[P_ /; BlockModQ[P]] := Module[**{**a = ProcBody[P]**}**,** {**StringReplace[PureDefinition[P], a–> "Procedure Body", 1], a**}**]**

In[2050] **:= Kr[x_, y_, z_] := Module[**{**a = x + y + z, b = 90**}**, b*a + a^2 + b]; PartProc[Kr]**
Out[2050]= {"Kr[x**_,** y**_,** z_]**:=** Module[{a= x+ y+ z**,** b= 90}**,**
Procedure Body]**", "**b*a+ a^2+ b"}

In[2054]**:= ReplaceProcBody[x_ /; BlockModQ[x], y_ /; StringQ[y]] := ToExpression[StringReplace[PureDefinition[x], ProcBody[x]–> y]]** In[2055]:= **ReplaceProcBody[Kr, "b*(x + y + z)"]; Definition[Kr]** Out[2055]= Kr[x**_,** y**_,** z_]:= Module[{a= x+ y+ z**,** b= 90}**,** b*****(x+ y+ z)]

A quite simple**ReplaceProcBody** function completes the previous fragment**;** the call**ReplaceProcBody[*x, y*]** returns*Null,* providing replacement of the body of a block or module***x*** by a new body***y*** that is given in string format. Furthermore, the updated object***x*** is activated in the current session. Both the**PartProc** procedure, and the**ReplaceProcBody** function are based on the above**ProcBody** procedure. Exactly the given circumstance provides a quite simple algorithm of these means.

Except the means considered in [28,30 -33] a number of means for operating with subprocedures is presented, here we will represent a useful procedure that analyzes the blocks/modules regarding presence in their definitions of subobjects of type {***Block,Module***}. The procedure call**SubsProcs[*x*]** returns generally the nested list of definitions in string format of all*subobjects* of the type {***Block, Module***} whose definitions are in the body of an object***x*** of type {***Block, Module***}. At that, the first sublist defines subobjects of***Module*–**type, the second sublist defines subobjects of***Block*–**type. In the presence of only one sublist the simple list is returned while in the presence of the***1*–**element simple list its element is returned. At lack of*subobjects* of the above type the call**SubsProcs[*x*]** returns the empty list, i.e. {} while on an object***x*,** different from a block or module, the call**SubsProcs[*x*]** is returned unevaluated. The following fragment represents source code of the**SubsProcs** procedure with the most typical examples of its usage.

In[2580]**:= SubsProcs[x_ /; BlockModQ[x]] := Module[**{**d, s =** {}**, g, k = 1, p, h = "", v = 1, R =** {}**, Res =** {}**, a = PureDefinition[x], j, m = 1, n = 0, b =** {**" := Module[**{**", " := Block[**{**"**}**, c = ProcBody[x]**}**,**

For[v, v <= 2, v++, If[StringFreeQ[c, b[[v]]], Break[], d = StringPosition[c, b[[v]]]]; For[k, k <= Length[d], k++, j = d[[k]][[2]]; While[m != n, p = StringTake[c, {**j, j**}**]; If[p == "[", m++; h = h <> p, If[p == "]", n++; h = h <> p, h = h <> p]]; j++]; AppendTo[Res, h]; m = 1; n = 0; h = ""]; Res = Map10[StringJoin, If[v == 1, " := Module[", " := Block["], Res]; g = Res;** {**Res, m, n, h**}**=** {{}**, 1, 0, "]"**}**; For[k = 1, k <= Length[d], k++, j = d[[k]][[1]]–2; While[m != n, p = StringTake[c,** {**j, j**}**]; If[p == "]", m++; h = p <> h, If[p == "[", n++; h = p <> h, h = p <> h]]; j—]; AppendTo[Res, h]; s = Append[s, j]; m = 1; n = 0; h = "]"]; Res = Map9[StringJoin, Res, g];** {**g, h**}**=**

{Res, ””}; Res = {}; For[k = 1, k <= Length[s], k++, For[j = s[[k]], j >= 1, j—, p = StringTake[c, {j, j}]; If[p == ” “, Break[], h = p <> h]]; AppendTo[Res, h]; h = ””]; AppendTo[R, Map9[StringJoin, Res, g]]; {Res, m, n, k, h, s}= {{}, 1, 0, 1, ””, {}}]; R = If[Length[R] == 2, R, Flatten[R]]; If[Length[R] == 1, R[[1]], R]]

In[2581] **:= P[x_, y_] := Module[{Art, Kr, Gs, Vg, a}, Art[c_, d_] := Module[{b}, c + d]; Vg[h_] := Block[{p = 90}, h^3 + p]; Kr[n_] := Module[{}, n^2]; Gs[z_] := Module[{}, x^3]; a = Art[x, y] + Kr[x*y]*Gs[x + y] + Vg[x*y]]**

In[2582] **:= P[90, 500]**
Out[2582]= 1 567 350 000 000 668
In[2583]**:= SubsProcs[P]**
Out[2583]= {{”Art[c_, d_]:= Module[{b}, c+ d]“, “Kr[n_]:= Module[{}, n^2]“,

“ Gs[z_]:= Module[{}, x^3]”}, {”Vg[h_]:= Block[{p= 90}, h^3+ p]”}} In[2584]**:= H[t_] := Module[{P}, P[h_] := Module[{a = 90}, a*h]; Cos[t]+ P[t]]** In[2585]**:= SubsProcs[H]**
Out[2585]= “P[h_]:= Module[{a= 90}, a*h]”

The **SubsProcs** procedure can be rather simply expanded, in particular, for determination of nesting levels of subprocedures, and also onto unnamed subprocedures. The**SubsProcs** procedure significantly uses also our means **BlockModQ, Map10, Map9, ProcBody, PureDefinition** considered above.

Moreover, in connection with the problem of nesting of blocks and modules essential enough distinction between definitions of the nested procedures in the systems**Mathematica** and**Maple** takes place. So, in the**Maple** system the definitions of subprocedures allow use of lists of the formal arguments identical with the main procedure containing them, whereas in the system **Mathematica** similar combination is inadmissible, causing in the course of evaluation of definition of the main procedure erroneous situations [30-33]. Generally speaking, the given circumstance causes certain inconveniences, demanding special attentiveness in process of programming of the nested procedures. In a certain measure the similar situation arises and in the case of crossing of lists of formal arguments of the main procedure and the local variables of its subprocedures whereas that is quite admissible in the**Maple** system [10-22,25-27]. In this context the**SubsProcs** procedure can be applied quite successfully and to the procedures containing subprocedures of type {***Block, Module***}, on condition of nonempty crossing of the list of the formal arguments of the main procedure along with the list of local variables of its subprocedures.

The following procedure provides return of the list of all blocks, functions and modules of the user packages uploaded into the current session, along with other active objects of the specified types. The next fragment represents source code of the**ProcsAct** procedure along with examples of its usage.

In[2526] **:= ProcsAct[] := Module[{a = Names[“*”], b = Names[“System`*”], c, d = {}, k = 1, j, h, t, g = {{“Module”}, {“Block”}, {“DynamicModule”}, {“Function”}, {“Others”}}},**

c = Select[a, ! MemberQ[b, #] &]; c = Select[c, ToString[Definition[#]] != “Null” && ToString[Definition[#]] != “Attributes[” <> ToString[#] <> “] = {Temporary}” && !

MemberQ[{ToString[#] <> " = {Temporary}", ToString[#] <> " = {Temporary}"}, ToString[Definition[#]]] &]; For[k, k <= Length[c], k++, h = c[[k]]; ClearAll[t]; Quiet[ProcQ1[Symbol[h], t]]; If[t === "Module", AppendTo[g[[1]], h],

If[t === "Block", AppendTo[g[[2]], h],
If[t === "DynamicModule", AppendTo[g[[3]], h],
If[QFunction[h], AppendTo[g[[4]], h], AppendTo[g[[5]], h]]]]]]; g]

In[2527] **:= ProcsAct[]**
Out[2527]= {{**"Module"**, "ActBFMuserQ", "ActCsProcFunc", "ActiveProcess", "ActRemObj", "Adrive", "Adrive1", "Affiliate", "Aobj", "Args", "ArgsBFM", "ArgsTypes", **…**},
{**"Block"**},
{**"DynamicModule"**},
{**"Function"**,"AssignL", "Attributes1", "AttributesH", "BinaryListQ", "BlockQ1", "ComplexQ", "ContextActQ", "ContextFromFile", "ContextQ", "CopyDir", **…**},
{**"Others"**, "AcNb", "ActUcontexts", "ClearOut", "CloseAll", "CsProcsFuncs", **…**}}

The procedure call **ProcsAct[]** returns the nested five-element list, sublists of which define by the*first* element the types of objects in the context {***"Block", "Module","DynamicModule","Function","Others"***} that are activated in the current session while other elements define names of objects corresponding to the*first* element of type. Meanwhile, it should be noted the*performance* of the**ProcsAct** procedure quite significantly depends on quantity of both the user means and the system means activated in the current session. Again it should be noted that in the***Mathematica***procedures*local* variables initially aren**'**t considered as undefinite**;** however, is possible to give them the status undefinite in the body of a procedure what visually illustrates the following rather transparent example, namely**:**

In[2547] **:= A[x___] := Module[{a, b, c}, b = {Attributes[a], Definition[a]}; ClearAll[a]; c = {Attributes[a], Definition[a]}; {b, c}]**
In[2548]**:= A[]**
Out[2548]= {{{Temporary}, Null}, {{}, Null]}}

Such reception is used and in the **ProcsAct** procedure, providing return of the type of an object***h*** through the*second* argument***t*** *–an undefinite variable–* at the call**ProcQ1[*h, t*].** In general, the**ProcsAct** procedure represents quite certain interest for certain appendices above all in procedural programming of problems of the system character.

The next fragment represents rather useful function **NamesProc,** whose call **NamesProc[]** returns the sorted list of names of the user modules, functions and blocks activated in the current session. In certain cases the**NamesProc** function can appear as a rather useful means. The next fragment represents source code of the**NamesProc** function with typical examples of its usage.

In[3617] **:= NamesProc[] := Select[Sort[Names["`*"]], Quiet[BlockFuncModQ[#]]&& ToString[Definition[#]] != "Null" && ToString[Definition[#]] != "Attributes[" <> ToString[#] <> "] = {Temporary}" && ! MemberQ[{ToString[#] <>" = {Temporary}",**

ToString[#] <> " = {Temporary}"}, ToString[Definition[#]]]&] In[3618]**:=**

NamesProc[]
Out[3618]= {A, Art, Df, F, G, H, Kr, NamesProc, ProcQ, Spos, Subs, Uprocs}

As one more example we will present the **Uprocs** procedure which is quite useful in the practical relation and also illustrates an approach to a certain expansion of the standard**Mathematica** means. The procedure call**Uprocs[]** returns simple or the nested list. In the first case in the current session the user procedures of any of***2*** types {***Block,Module***} have been not activated, while in the second case the list elements returned by the**Uprocs** procedure are***are***element sublists whose first elements define names of the user blocks/ modules activated in the current session, the second define their headings in string format, the third elements define type of procedures {***Block|Module***}. The following fragment represents source code of the**Uprocs** procedure and the most typical example of its usage.

In[2448] := **Gs[x_] := Block[{a, b, c}, Evaluate[(a*x + x^b)/c]]**
In[2449]:= **S[x_] := Block[{y = a, h = b}, G[Pi/2, y*x]]**
In[2450]:= **S[x_] := Module[{y = a, h = b}, G[Pi/2, y*x]]**
In[2451]:= **S[x_, y_] := Block[{z = a, h = b}, G[Pi/2, (y*x)/z]]**
In[2452]:= **Bl[y_] := Block[{h = z}, G[Pi/2, y]]**
In[2453]:= **MM[x_, y_] := Module[{}, x + y]**

In[2454]:= **Uprocs[] := Module[{a, b, c, d, h, g, k, t1, t2},**
a := "_$Art25_Kr18$_.txt"; {c, g}= {{}, {}}; Save[a, "`*"]; b := Map[ToString, Flatten[DeleteDuplicates[ReadList[a, String]]]];

For[k = 1, k <= Length[b], If[StringCount[First[b[[{k}]]], " := Module[{"] != 0 && StringTake[First[b[[{k}]]], {1}] != " " || StringCount[First[b[[{k}]]], " := Block[{"] != 0 && StringTake[First[b[[{k}]]], {1}] != " ", AppendTo[c, First[b[[{k}]]]], Null]; k = k + 1]; For[k = 1, k <= Length[c], d = Quiet[First[c[[{k}]]]]; h = Quiet[Symbol[StringTake[d, First[First[StringPosition[d, "["]]]–1]]]; t1 = If[StringCount[d, " := Module[{"] != 0, Module, Block];

t2 = Quiet[StringTake[d, Last[First[StringPosition[d, "]"]]]]]; If[BlockModQ[h], AppendTo[g, {h, t2, t1}], Null]; k = k + 1]; DeleteFile[a]; g] In[2455]:= **Uprocs[]**
Out[2455]= {{Bl, "Bl[y_]", Block}, {Gs, "Gs[x_]", Block}, {H, "H[t_]", Module},

{P, "P[x_, y_]", Module}, {MM, "MM[x_, y_]", Module}}

The procedure call **ExtrCall[*z, y*]** returns*True* if the user block, function or module***y*** contains the calls of a block/function/module***z,*** otherwise*False* is returned. If the call as an argument***z*** defines the list of names of blocks/ functions/modules, the sublist of names from***z*** of blocks/functions/modules whose calls enter into an object***y*** is returned. In the case if the first optional argument***z*** is absent, then the call**ExtrCall[*y*]** returns the list of the system means whose calls enter into definition of the user function, block, module ***y.*** The following fragment represents source code of the**ExtrCall** procedure along with the most typical examples of its usage.

In[2547]:= **ExtrCall[z___, y_ /; BlockFuncModQ[y]] := Module[{b, p, g, x, a = Join[CharacterRange["A", "Z"], CharacterRange["a", "z"]]},**

If[{z}== {}, p = PureDefinition[y]; If[ListQ[p], Return[$Failed]]; g =

ExtrVarsOfStr[p, 2];
g = Select[g = Map[" " <> # <> "[" &, g], ! StringFreeQ[p, #] &];
g = Select[Map[If[SystemQ[p =StringTake[#, {2,–2}]], p]&, g], ! SameQ[#, Null] &];
If[Length[g] == 1, g[[1]], g], b[x_] := Module[{c = DefFunc3[ToString[y]], d, h, k = 1, t = {}}, h = StringPosition[c, ToString[x] <> "["]; If[h == {}, Return[False], d = Map[First, h]; For[k, k <= Length[d], k++, AppendTo[t, If[! MemberQ[a, StringTake[c, {d[[k]]–1, d[[k]]–1}]], True, False]]]]; t[[1]]]; If[! ListQ[z], b[z], Select[z, b[#] &]]]]

In[2548] **:= Map3[ExtrCall, Run, {Attrib, SearchDir, SearchFile, Df, Uprocs}]**
Out[2548]= {True, True, True, False, False}
In[2549]**:= ExtrCall[{Run, Write, Read, If, Return}, Attrib]**
Out[2549]= {Run, Read, If, Return}
In[2550]**:= Map[ExtrCall, {BlockFuncModQ, ExtrCall}]**
Out[2550]= {{"Flatten", "FromCharacterCode", "If", "Module", "StringTake", "StringReplace"}, {"Append", "CharacterRange", "For", "If", "Join", "Length", "Module", "Return", "Select", "StringJoin", "StringPosition", "StringTake"}} The definition of the**ExtrCall** procedure along with the*standard* means uses a number of our tools such as**BlockFuncModQ, PureDefinition, SystemQ, DefFunc3** and**ExtrVarsOfStr** which are considered in the present book and in [30,33]. The**ExtrCall** procedure has a series of useful enough appendices, first of all, in the problems of system character. Meanwhile, it must be kept in mind that the**ExtrCall** procedure correctly processes only unique objects, but not objects of the same name by returning on them**$Failed.**

In addition to earlier presented**TestArgsTypes** procedure providing the call of a specified block, function, module in such manner that returns result of this procedure call in the absence of inadmissible actual arguments or the list consisting off values {***True,False***} whose order corresponds to an order of the actual arguments at a call of the tested object of the specified type the **TestProcCalls** procedure is of a certain interest. The call**TestProcCalls[*x,y*]** returns the nested list whose elements have format {***j,"n",****True*|*False*} where ***j*** **–** the ordinal number of a formal argument,***"n"*** **–** a formal argument in the string format, {*True*|*False*}**–** value which determines admissibility*(True)* or inadmissibility*(False)* of an actual value determined by a list***y*** and received by a formal argument {***j, n***} in a point of the call of an object***x*.** Furthermore, it is supposed that an object***x*** defines the*fixed* number of formal arguments and*lengths* of lists defining formal arguments and***y*** are identical, otherwise the procedure call returns**$Failed.**

In[5057]**:= TestProcCalls[x_ /; BlockFuncModQ[x], y_ /; ListQ[y]] :=**

Module[{d, p, a = Args[x], b = {}, r, c = "_ /; ", k = 1, v}, a = Map[ToString1, If[NestListQ[a], a[[1]], a]]; If[Length[a] != Length[y] || MemberQ[Map[! StringFreeQ[#, "__"] &, a], True], $Failed, v = If[NestListQ[v = Args[x, 90]], v[[1]], v]; For[k, k <= Length[a], k++, p = a[[k]]; AppendTo[b, If[StringTake[p, {–1,–1}] == "_", True, If[! StringFreeQ[p, c], d = StringSplit[p, c]; r = ToExpression[d[[1]]]; {ToExpression[{d[[1]] <> "=" <> ToString1[y[[k]]], d[[2]]}][[2]], ToExpression[d[[1]] <> "=" <> ToString[r]]}[[1]], d = StringSplit[p, "_"]; ToString[Head[y[[k]]]] == d[[2]]]]]]]; {k, d}= {1, Partition[Riffle[v, b], 2]}; While[k <= Length[d], PrependTo[d[[k]], k]; k++]; d]] In[5058]**:= TestProcCalls[SuffPref,**

{**“IAN_RANS_RAC_90_73”, “90_73”, 2**}**]** Out[5058]= {{1, “S“, True}, {2, “s“, True}, {3, “n“, True}}

In[5059] **:= TestProcCalls[SuffPref,** {**“IAN_RANS_RAC_90_73”, 50.90, 7.3**}**]**
Out[5059]= {{1, “S“, True}, {2, “s“, False}, {3, “n“, False}}
In[5060]**:= F[x_String, y_ /; IntegerQ[y]] :=** {**x, y**}**; TestProcCalls[F,** {**6, “avz”**}**]**
Out[5060]= {{1, “x“, False}, {2, “y“, False}}

The previous fragment presents source code of the **TestProcCalls** procedure with examples of its usage. The procedure call**TestProcCalls[*x*]** successfully processes the unique objects and the objects of the same name, at that in the *second* case the*first* subobject from the list returned by the call**Definition[*x*]** is processed. At checking of values on admissibility in the case of a formal argument of format***“arg_/;Test(arg)”*** is required previously to calculate***arg*** and only then to check a logical value***Test(arg).*** However this operation in body of the**TestProcCalls** procedure updates***arg*** outside of the procedure what in general is inadmissible. Therefore for elimination of this situation a quite simple reception*(that can be easily seen from the presented procedure code)* without redefinition of global variables of the current session that have the same name with formal arguments of the tested object***x*** of the type stated above has been used. This approach to the organization of algorithm of the procedure quite answers the concept of the robust programming. At that, the**TestProcCalls** procedure allows a series of modifications useful enough for procedural programming in the**Mathematica** system.

In contrast to the previous procedures the next **ProcActCallsQ** procedure tests existence in the user block, function or module x the existence of calls of the user means active in the current session that are provided by usages. The procedure call**ProcActCallsQ[*x*]** returns*True* if definition of a module, block, function***x*** contains the calls of tools of the similar type, otherwise*False* is returned. Moreover, thru the second optional argument***y*** *–an undefinite variable*– the procedure call**ProcActCallsQ[*x*, *y*]** returns the list of the user software whose calls are in definition of a block, function or a module***x.***

In[5070] **:= ProcActCallsQ[x_ /; BlockFuncModQ[x], y___] := Module[**{**a, b, c =** {}**, d, k = 1, h = “::usage = “**}**, Save[b = “Art26$Kr18”, x]; For[k, k < Infinity, k++, d = Read[b, String]; If[SameQ[d, EndOfFile], Break[], If[! StringFreeQ[d, h], AppendTo[c, StringSplit[StringTake[d,** {**1, Flatten[StringPosition[d, h]][[1]]–1**}**], ” /: “][[1]]]]]]; DeleteFile[Close[b]]; c = Select[c, SymbolQ[#] &]; b = If[MemberQ[c, ToString[x]], Drop[c, 1], c]; If[**{**y**}**!=** {}**&& ! HowAct[**{**y**}**[[1]]],** {**y**}**=** {**b**}**]; If[b ==** {}**, False, True]]**

In[5071] **:=** {**ProcActCallsQ[ProcQ, h], h**}
Out[5071]= {True, {”SymbolQ“, “SystemQ“, “UnevaluatedQ“, “ToString1“, “StrDelEnds“, “SuffPref“, “ListStrToStr“, “Definition2“, “HowAct“, “Mapp“, “SysFuncQ“, “Sequences“, “Contexts1“, “ClearAllAttributes“, “SubsDel“, “HeadPF“, “BlockFuncModQ“, “PureDefinition“, “Map3“, “MinusList”}}
In[5072]**:=** {**ProcActCallsQ[ToString1, s], s**}
Out[5072]= {True, {”StrDelEnds“, “SuffPref”}}
In[5073]**:= G[x_String, y_ /; ! HowAct[y]] := If[StringLength[x] == 90, y = x, y = x <> “500”];** {**ProcActCallsQ[G, Gs], Gs**}
Out[5073]= {True, {”HowAct”}}

In[5074]:= {**ProcActCallsQ[StrStr, Sv], Sv**}
Out[5074]= {False, {}}
In[5075]:= {**ProcActCallsQ[ProcActCallsQ, Gsv], Gsv**}
Out[5075]= {True, {"BlockFuncModQ", "PureDefinition", "UnevaluatedQ", "SymbolQ", "ToString1", "StrDelEnds", "SuffPref", "ListStrToStr", "Definition2", "HowAct", "SystemQ", "Mapp", "ProcQ", "ClearAllAttributes", "SubsDel", "Sequences", "HeadPF", "SysFuncQ", "Contexts1", "Map3", "MinusList"}}
In[5076]:= **F[x_] := If[NestListQ[x], x, ToString1[x]]**
In[5077]:= {**ProcActCallsQ[F, v], v**}
Out[5077]= {True, {"NestListQ", "ToString1", "StrDelEnds", "SuffPref"}}

The previous fragment gives source code of the **ProcActCallsQ** procedure along with some typical examples of its usage. The procedure is of interest at the structural analysis of the user blocks/functions/modules; furthermore, the exhaustive analysis belongs only to the user means active in the current session of the system and provided by the standard usages.
In certain cases the question of definition of all tools used by the user block, function, module that are activated in the current session including means for which the usages are missing represents a certain interest. This problem is solved by the**ProcContent** procedure which provides the analysis of an activated object***x*** of the above type with a correct heading, concerning the existence in its definition of the user means both internal, and external, that are supplied with an usage or without it. The procedure call**ProcContent[*x*]** returns the nested***3***–element list whose first element defines the name of a block/function/module***x*,** the second element defines the list of names of all external blocks, functions or modules used by the object***x*** whereas the third element defines the list of names of the internal blocks, functions or modules defined in the body of***x*.** The following fragment represents source code of the**ProcContent** procedure along with typical examples of its usage.

In[5080] := **Kr[x_, y_] := Plus[x, y]; Art[x_] := Module[{a = 90, b = 500, c = ToString1[x], d, g}, c = Kr[a, b]; d[y_] := Module[{}, y]; g[z_] := Block[{}, z + 90]; c]; V[x_] := Module[{c = StrStr[x], d, g}, G[a, b]; d[y_] := Module[{}, y]; g[z_] := Block[{}, z]; c]**

In[5081]:= **ProcContent[x_ /; BlockFuncModQ[x]] := Module[{a, f, b = SubProcs[x][[1]]}, f[y_] := Module[{a1 = "$Art2618Kr$", b1 = "", c = {y}, d, h = "", p},**

Save[a1, y]; While[! SameQ[b1, EndOfFile], b1 = Read[a1, String]; If[! MemberQ[{" ", "EndOfFile"}, ToString[b1]], h =h <> ToString[b1]; Continue[], d = Flatten[StringPosition[h, " := ", 1]]]; If[d == {}, h = ""; Continue[], p = StringTake[h, {1, d[[1]]–1}]; If[! SameQ[Quiet[ToExpression[p]], $Failed], AppendTo[c, StringTake[p, {1, Flatten[StringPosition[p, "[", 1]][[1]]–1}]]; h = "", Null]]]; a1 = Map[ToExpression, {DeleteFile[Close[a1]], c}[[2]]]; DeleteDuplicates[a1]]; a = f[x]; {x, If[Length[a] > 1, a[[2 ;;–1]], {}], If[Length[b] > 1, Map[ToExpression, Map[HeadName, b[[2 ;;–1]]]], {}]}]

In[5082] := **ProcContent[V]**
Out[5082]= {V, {StrStr, G, HowAct}, {d, g}}
In[5083]:= **ProcContent[Art]**

Out[5083]= {Art, {ToString1, StrDelEnds, SuffPref, Kr}, {d, g}} In[5084]:= **ProcContent[ToString1]**
Out[5084]= {ToString1, {StrDelEnds, SuffPref}, {}}}
In[5085]:= **V[x_] := Module[{a = 5, b = 6, c, d, g, Gt}, c = Gt[a, b]; d[y_] := Module[{}, y]; g[z_] := Block[{a = 6, b =9}, z/73]; ToString1[x] <> StrStr[x]]**

In[5086]:= **ProcContent[V]**
Out[5086]= {V, {ToString1, StrDelEnds, SuffPref, StrStr}, {d, g}}

At that, the **ProcContent** procedure along with standard functions enough essentially uses the procedures**BlockFuncModQ, SubProcs** together with a simple function, whose call**HeadName[*x*]** returns the name of a heading***x*** in string format. These means were considered in the present book above.

The function call **ProcFuncCS[]** returns the nested three-element list whose sublists define names in string format according of the user blocks,modules and functions, whose definitions were evaluated in the current session. The next fragment represents source code of the**ProcFuncCS** function together with a typical example of its usage.

In[2532]:= **ProcFuncCS[] := Quiet[Map3[Select,Names["`*"], {BlockQ[#] &, FunctionQ[#] &, ModuleQ[Symbol[#]] &}]]**

In[2533] **:= G[x_String, y_ /; ! HowAct[y]] := If[StringLength[x] == 90, y = x, y = x <> "500"]; GS[x_] := Block[{a = 90, b = 500}, x]; F[x_] := If[NestListQ[x], x, ToString1[x]]; GG[y_] := Module[{a = 90, b = 500, c = 2015, d = {42, 47, 67}}, y]; ProcFuncCS[]** Out[2533]= {{"GS"}, {"F", "G"}, {"GG"}}

The operator **HeadCompose[*a, b, c, d*]** which was in the previous releases of the system*(now the operator isn't documented)* returns the composition of the identifiers in the form given below, namely**:**

In[2545] **:= HeadCompose[G, x, y, z]**
Out[2545]= G[x][y][z]
Such form, for example, can be useful in various functional transformations. The given operator can be useful enough also at the organization of the user functions, allowing to transfer in quality of the*actual* values for their formal arguments the headings of functions along with their formal arguments. At the same time, this tool in general doesn't represent an especial interest what induced its bringing outside the system. On the other hand, it is possible to represent a certain analog of this tool which has significantly larger applied interest, namely the**FunCompose** procedure whose call**FunCompose[*L, x*]** allows to create the nested functions from the list***L*** of functions, modules or blocks from an expression given by its second argument***x*.** The following a quite simple fragment rather visually illustrates the aforesaid.

In[2551]:= **FunCompose[t_ /; ListQ[t], x_] := Module[{a, k = 2}, a = t[[1]]@x; For[k, k <= Length[t], k++, a = t[[k]]@a]; a]**

In[2552] **:= FunCompose[{F, G, H, T, W, Q, V, U}, Sin[z]]**
Out[2552]= F[G[H[T[W[Q[V[U[Sin[z]]]]]]]]]
In[2553]:= {**FunCompose[{Sin, Cos, Log}, 9.42], FunCompose[{Sin, Cos,**

Tan, Sqrt}, 500.90]}

Out[2553]= {–0.0000114144, 0.786906}

For organization of transfer of identifiers of functions as the actual values it is possible to use constructions, for example, of the following rather simple formats, namely:
In[2555]:= **F[x_] := x^3; SV[z_] := F@z + z^3; VSV[Id_, z_] :=**

Module[{}, Id@(z^2 + 6)]; {VSV[F, h], SV[45]}
Out[2555]= {(6+ h^2)^3, 182250}
along with a number of similar useful enough constructions.
For temporary removal from the current session of the**Mathematica** system of the user blocks, functions or modules quite useful**DelRestPF** procedure serves whose source code along with typical examples of usage represents the following fragment.

In[2579]:= **F[x_] := x^3; SV[z_] := F@z + z^3; VSV[Id_, z_] := Module[{}, Id@(z^2 + 6)]; F[x_, y_] := x + y; SetAttributes[F, {Protected, Listable}]; SetAttributes[SV, Listable]**

In[2580] := **DelRestPF[r_ /; MemberQ[{"d", "r"}, r], x___] := Module[{b, c, p, f = "$Art26Kr18$.mx", a = Quiet[Select[{x}, BlockFuncModQ[#] &]], k = 1}, If[r == "d", b = Map[Definition2, a]; Save[f, b]; Map[ClearAllAttributes, a]; Map[Remove, a];, c = Get[f]; DeleteFile[f]; For[k, k <= Length[c], k++, p = c[[k]]; ToExpression[p[[1 ;;–2]]]; ToExpression["SetAttributes[" <> StringTake[p[[1]], {1, Flatten[StringPosition[p[[1]], "["]][[1]]–1}] <> "," <> ToString[p[[–1]]] <> "]"]]]]**
In[2581]:= **DelRestPF["d", F, SV, VSV]**
In[2582]:= **Map[Definition2, {F, SV, VSV}]**

Out[2582] = {Definition2[F], Definition2[SV], Definition2[VSV]} In[2583]:= **DelRestPF["r"]**
In[2584]:= **Map[Definition2, {F, SV, VSV}]**
Out[2584]= {{"F[x_]:= x^3", "F[x_, y_]:= x+ y", {Listable, Protected}},

{"SV[z_]:= F[z]+ z^3", {Listable}},
{"VSV[Id_, z_]:= Module[{}, Id[z^2+ 6]]", {}}}

In[2585] := **DelRestPF1[r_ /; MemberQ[{"d", "r"}, r], f_/; StringQ[f], x___] := Module[{a =Quiet[Select[{x}, BlockFuncModQ[#] &]], b, c, p, k = 1}, If[r == "d", b = Map[Definition2, a]; Save[f, b]; Map[ClearAllAttributes, a]; Map[Remove, a];, c = Get[f]; DeleteFile[f]; For[k, k <= Length[c], k++, p = c[[k]]; ToExpression[p[[1 ;;–2]]]; ToExpression["SetAttributes[" <> StringTake[p[[1]], {1, Flatten[StringPosition[p[[1]], "["]][[1]]–1}] <> "," <> ToString[p[[–1]]] <> "]"]]]]**

In[2586] := **DelRestPF1["d", "C:\Temp\Tallinn", F, SV, VSV]** In[2587]:= **Map[Definition2, {F, SV, VSV}]**
Out[2587]= {Definition2[F], Definition2[SV], Definition2[VSV]} In[2588]:= **DelRestPF1["r", "C:\Temp\Tallinn"]**
In[2589]:= **Map[Definition2, {F, SV, VSV}]**
Out[2589]= {{"F[x_]:= x^3", "F[x_, y_]:= x+ y", {Listable, Protected}},

{"SV[z_]:= F[z]+ z^3", {Listable}},
{"VSV[Id_, z_]:= Module[{}, Id[z^2+ 6]]", {}}}

The procedure call **DelRestPF[*"d"*, *x*, *y*, …]** returns*Null*, i.e.*nothing*, deleting from the

current session the user blocks, functions and/or modules {***x, y, …***} while the subsequent call**DelRestPF[*"r"*]** returns*Null,* i.e.*nothing*, restoring their availability in the current session or in other session with preservation of the options and attributes ascribed to them. The procedure is quite useful in a number of applications, first of all, of system character. Moreover, the procedure is oriented onto work only with one list of objects, creating only a fixed file with the saved objects. Meanwhile, a very simple modification of the procedure provides its expansion onto any number of lists of the user blocks, functions and modules, allowing temporarily to delete them at any moments from the current session with the subsequent their restoration in the current session or other session of the system. So, the previous fragment is completed by one of similar useful modifications which has a number of useful appendices of the system character.
The procedure call**DelRestPF1[*"d",w, x, y, z,…*]** returns*Null,* i.e. nothing, deleting from the current session the user blocks, functions and/or modules {***x, y, z, …***} with saving of them in a datafile***w,*** whereas the subsequent call **DelRestPF1[*"r",w*]** returns*Null,* i.e. nothing, restoring their availability in the current session or in other session from the datafile***w*** with preservation of the options and attributes ascribed to them.

The built **–**in***Math*–**language for programming of the branching algorithms along with the**"If"** offer allows use of unconditional transitions on the basis of the**Goto** function which is encoded in the form**Goto[*h*],** unconditionally passing control into a point defined by the construction**Label[*h*].** As a rule, the**Goto** function is used in procedural constructions, however unlike the built**–**in***goto*–**function of the**Maple** system it can be used also in the input constructions of the**Mathematica** system. Moreover, as a**Label** any correct expression is allowed, including also sequence of expressions whose the last expression defines actually label**;** at that, the**Label** concerning a module can be both the global variable, and the local variable. Meanwhile, in order to avoid possible misunderstandings, the**Label** is recommended to be defined as a local variable because the global**Label** calculated outside of a module is always acceptable for the module, however calculated in the module body quite can distort calculations outside of the module. At that, multiplicity of occurrences of identical**Goto–**functions into a procedure is quite naturally and is defined by the realized algorithm while with the corresponding tags **Label** the similar situation, generally speaking, is inadmissible**;** at that, it is not recognized at evaluation of a procedure definition and even at a stage of its performance, often substantially distorting the planned task algorithm. In this case only point of a module body which is marked by the first such **Label** receives the control. Moreover, it must be kept in mind that lack of a **Label[*a*]** for the corresponding call**Goto[*a*]** in a block or module at a stage of evaluation of its definitions isn**'**t recognized, however only at the time of performance with the real appeal to such**Goto[*a*].** The interesting examples illustrating the told can be found in our books [28-33].

In this connection the **GotoLabel** procedure can represent a certain interest whose call**GotoLabel[*P*]** allows to analyse a procedure***P*** on the subject of formal correctness of use of**Goto-**functions and**Label** tags corresponding to them. The procedure call**GotoLabel[*P*]** returns the nested***3*–**element list whose first element defines the list of all**Goto-**functions used by a module***P,*** the second element defines the list of all tags*(without their multiplicity)***,** the third element defines the list, whose sublists define**Goto-**functions with the tags corresponding to them*(at that,as the first elements of these sublists the calls* *of the***Goto**-*functions appear,whereas multiplicities of functions and tags remain)***.** The following fragment represents source code of the**GotoLabel** procedure along with typical examples of its usage.

In[2540] **:= GotoLabel[x_ /; BlockModQ[x]] := Module[{b, c = {{}, {}, {}}, d, p, a = Flatten[{PureDefinition[x]}][[1]], k = 1, j, h, v = {}, t}, b = ExtrVarsOfStr[a, 1]; b = DeleteDuplicates[Select[b, MemberQ[{"Label", "Goto"},#] &]]; If[b == {}, c, d = StringPosition[a, Map[" " <> # <> "[" &, {"Label", "Goto"}]]; t = StringLength[a]; For[k, k <= Length[d], k++, p = d[[k]]; h = ""; j = p[[2]]; While[j <= t, h = h <> StringTake[a, {j, j}]; If[StringCount[h, "["] == StringCount[h, "]"], AppendTo[v, StringTake[a, {p[[1]] + 1, p[[2]]–1}] <> h]; Break[]]; j++]]; h = DeleteDuplicates[v]; {Select[h, SuffPref[#, "Goto", 1] &], Select[h, SuffPref[#, "Label", 1] &], Gather[Sort[v], #1 == StringReplace[#2, "Label["–> "Goto[", 1] &]}]]**

In[2541]**:= ArtKr[x_ /; IntegerQ[x]] := Module[{prime, agn}, If[PrimeQ[x], Goto[9; prime], If[OddQ[x], Goto[agn], Goto[Sin]]]; Label[9; prime]; Print[x^2]; Goto[Sin]; Print[NextPrime[x]]; Goto[Sin]; Label[9; prime]; Null]** In[2542]:= **Kr[x_ /; IntegerQ[x]] := Module[{prime, agn, y},**
If[PrimeQ[x], Goto[prime], If[OddQ[x], Goto[agn], Goto[agn]]];

Label[9; prime]; y = x^2; Goto[agn]; Label[agn]; y = NextPrime[x]; Label[agn]; y]
In[2543]**:= GotoLabel[ArtKr]**
Out[2543]= {{"Goto[9;prime]", "Goto[agn]", "Goto[Sin]"},{"Label[9;prime]"},

{{ "Goto[9; prime]", "Label[9; prime]","Label[9; prime]"}, {"Goto[agn]"}, {"Goto[Sin]", "Goto[Sin]", "Goto[Sin]"}}} In[2544]**:= GotoLabel[Kr]**
Out[2544]= {{"Goto[prime]", "Goto[agn]"}, {"Label[9; prime]", "Label[agn]"},

{{ "Goto[agn]", "Goto[agn]", "Goto[agn]", "Label[agn]", "Label[agn]"}, {"Goto[prime]"}, {"Label[9; prime]"}}} In[2545]**:= Map[GotoLabel, {GotoLabel, TestArgsTypes, }]**
Out[2545]= {{{}, {}, {}}, {{}, {}, {}}}
In[2546]**:= Map[GotoLabel, {SearchDir, StrDelEnds, OP, BootDrive}]** Out[2546]= {{{}, {}, {}}, {{}, {}, {}}, {{"Goto[ArtKr]"}, {"Label[ArtKr]"},

{{"Goto[ArtKr]", "Label[ArtKr]"}}}, {{"Goto[avz]"}, {"Label[avz]"}, {{"Goto[avz]", "Goto[avz]", "Goto[avz]", "Label[avz]"}}}}

We will note that existence of a nested list with the third sublist containing **Goto**–functions without tags corresponding to them, in the result returned by a call**GotoLabel[*P*]** not necessarily speaks about existence of the function calls**Goto[*x*]** for which not exists a tag**Label[*x*].** It can be, for example, in the case of generation of a value depending on some condition.

In[2550] **:= Av[x_Integer, y_Integer, p_ /; MemberQ[{1, 2, 3}, p]] := Module[{}, Goto[p]; Label[1]; Return[x + y];**
Label[2]; Return[N[x/y]];
Label[3]; Return[x*y]]

In[2551] **:= Map[Av[500, 90, #] &, {1, 2, 3, 4}]**
Out[2551]= {590, 5.55556, 45000, Av[500, 90, 4]}

In[2552]**:= GotoLabel[Av]**
Out[2552]= {{"Goto[p]"}**,** {"Label[1]"**,** "Label[2]"**,** "Label[3]"}**,** {{"Goto[p]"}**,**

{"Label[1]"}**,** {"Label[2]"}**,** {"Label[3]"}}}

For example, according to simple example of the previous fragment the call **GotoLabel[*Av*]** contains {"Goto[p]"} in the*third* sublist what, at first sight, it would be possible to consider as a certain impropriety of the corresponding call of**Goto-**function. However, all the matter is that a value of the actual***p***argument in the call***Av*[*x, y, p*]** and defines a tag, really existing in definition of this procedure, i.e. a**Label[*p*].** Thus, the**GotoLabel** procedure only at the formal level analyzes existence of**Goto-**functions,**"*incorrect*"** from its point of view along with**"*excess*"** tags. Whereas refinement of the results received on the basis of a call**GotoLabel[*P*]** lies on the user, first of all, by means of analysis of accordance of source code of a***P*** procedure to the correctness of the required algorithm.

The structured paradigm of programming doesn 't assume use in programs of the***goto-***constructions allowing to transfer control from bottom to top. At the same time, in a number of cases the use of**Goto–**function is effective, in particular, at needing of embedding into the**Mathematica** environment of a program which uses unconditional transitions on the basis of the***goto-***offer. For example,**Fortran–**programs can be adduced as a quite typical example that are very widespread in the scientific appendices. From our experience follows, that the use of**Goto–**function allowed significantly to simplify the embedding into the**Mathematica** environment of a number of rather large **Fortran–**programs relating to engineering and physical applications which very widely use the***goto–***constructions. Right there it should be noted that from our standpoint the**Goto**-function of the**Mathematica** system is more preferable, than***goto–***function of the**Maple** system in respect of efficiency in the light of application in*procedural* programming of various appendices, including appendices of the system character.

As it was already noted, the **Mathematica** allows existence of the objects of the same name with various headings which identify objects, but not their names. The standard**Definition** function and our procedures**Definition2, PureDefinition,** and others by name of an object allow to receive definitions of all active subobjects in the current session with identical names, but with various headings. Therefore there is quite specific problem of removal from the current**Mathematica** session not of all objects with a concrete name, but only subobjects with concrete headings.
The**RemovePF** procedure solves this problem**;** its call**RemovePF[*x*]** returns *Null***,** i.e. nothing, providing removal from the current session of the objects with headings***x*** which are determined by the factual argument***x****(a heading in string format or their list)***.** In the case of the incorrect headings determined by an argument***x*,** the call**RemovePF[*x*]** is returned unevaluated. The given procedure is quite useful in procedural programming. The fragment below represents source code of the**RemovePF** procedure along with examples of its typical usage for removal of subobjects at the objects of the same name.

In[2620]**:= RemovePF[x_ /; HeadingQ1[x] || ListQ[x] && DeleteDuplicates[Map[HeadingQ1, x]] == {True}] :=**

Module[{b, c = {}, d, p, k = 1, j, a = DeleteDuplicates[Map[HeadName,

Flatten[{x}]]]}, b =Map[If[UnevaluatedQ[Definition2, #], {“90”,{}}, Definition2[#]]&, a]; For[k, k <= Length[a], k++, p = b[[k]]; AppendTo[c, Select[Flatten[{p[[1 ;;–2]], “SetAttributes[” <> a[[k]] <> “, ” <> ToString[p[[–1]]] <> “]”}], ! SuffPref[#, x, 1] &]]]; Map[ClearAllAttributes, a]; Map[Remove, a]; Map[ToExpression, c]; a = Definition2[b = HeadName[x]]; If[a[[1]] === “Undefined”, ToExpression[“ClearAttributes[” <> b <> “,” <> ToString[a[[2]]] <> “]”], Null]]]

In[2621] **:= M[x_ /; SameQ[x, “avz”], y_] := Module[{a, b, c}, y]; F[x_, y_Integer] := x + y; F[x_, y_] := x + y; F[x_, y_, z_] := x + y + z; M[x_ /; x == “avz”] := Module[{a, b, c}, x]; M[x_, y_, z_] := x + y + z;**

M[x_ /; IntegerQ[x], y_String] := Module[{a, b, c}, x]; M[x_, y_] := Module[{a, b, c}, “agn”; x + y]; M[x_String] := x; M[x_ /; ListQ[x], y_] := Block[{a, b, c}, “agn”; Length[x] + y];

SetAttributes[M, Protected]; SetAttributes[F, Listable] In[2622]**:= Definition[M]**
Out[2622]= Attributes[M]= {Protected}

“ M[x_ /; x=== "avz", y_]:= Module[{a, b, c}, y]” “M[x_ /; x== "avz"]:= Module[{a, b, c}, x]”
“M[x_, y_, z_]:= x+ y+ z”
“M[x_ /; IntegerQ[x], y_String]:= Module[{a, b, c}, x]” “M[x_ /; ListQ[x], y_]:= Block[{a, b, c}, "agn"; Length[x]+ y]” “M[x_, y_]:= Module[{a, b, c}, "agn"; x+ y]” “M[x_String]:= x”

In[2623]:= **Definition[F]**

Out[2623] = Attributes[F]= {Listable}
“F[x_, y_Integer]:= x+ y”
“F[x_, y_]:= x+ y”
“F[x_, y_, z_]:= x+ y+ z”

In[2624] **:= RemovePF[{“M[x_, y_]”, “F[x_, y_, z_]”, “M[x_String]”, “M[x_, y_, z_]””, “F[x_, y_Integer]”, “v[t_]”}]**
In[2625]:= **Definition[M]**
Out[2625]= Attributes[M]= {Protected}
“M[x_ /; x=== "avz", y_]:= Module[{a, b, c}, y]” “M[x_ /; x== "avz"]:= Module[{a, b, c}, x]”
“M[x_ /; IntegerQ[x], y_String]:= Module[{a, b, c}, x]” “M[x_ /; ListQ[x], y_]:= Block[{a, b, c}, "agn"; Length[x]+ y]”
In[2626]:= **Definition[F]**
Out[2626]= Attributes[F]= {Listable}
“F[x_, y_]:= x+ y”
In[2627]:= **Definition[F]**
Out[2627]= Null

For ensuring of correct uploading of the user block/function/module***x*** in the current session on condition of possible need of additional reloading in the current session also of non-standard blocks/functions/modules whose calls are used in such object***x*,** the**CallsInProc** procedure can be useful enough, whose call**CallsInProc[*x*]** returns the list of all standard functions, external and internal blocks/functions/modules, whose calls are used by an object*x* of the specified type. The following fragment represents source code of the **CallsInProc** procedure along with the typical examples of its usage.

```
In[2660] := CallsInProc[P_ /; BlockFuncModQ[P]] := Module[{b, c ={}, k = 1, a =
ToString[FullDefinition[P]], TN}, TN[S_/; StringQ[S], L_ /; ListQ[L] &&

Length[Select[L, IntegerQ[#] &]] ==Length[L] && L != {}] := Module[{a1 = ””, c1,
b1 = {}, k1, p = 1}, For[p, p <= Length[L], p++, For[k1 = L[[p]]–1, k1 != 0, k1—, c1 =
StringTake[S, {k1, k1}]; a1 = c1 <> a1; If[c1 === ” “, a1 = StringTake[a1, {2,–1}];
If[Quiet[Check[Symbol[a1], False]] === False, a1 = ””; Break[], AppendTo[b1, a1];
a1 = ””; Break[]]]]]; b1]; b = TN[a, b = DeleteDuplicates[Flatten[StringPosition[a,
“[“]]]][[2 ;;–1]]; b = Sort[DeleteDuplicates[Select[b, StringFreeQ[#, “`”] && !
MemberQ[{“Block”, ToString[P], “Module”}, #] && ToString[Definition[#]] !=
“Null” &]]]; k = Select[b, SystemQ[#] &]; c = MinusList[b, Flatten[{k,
ToString[P]}]]; {k, c, DeleteDuplicates[Map[Context, c]]}]

In[2661]:= CallsInProc[StringDependQ]
Out[2661]= {{”Attributes“, “Flatten“, “If“, “Length“, “ListQ“, “Select“,

“ StringFreeQ“, “StringQ”}, {”HowAct“, “ListStrQ”}, {”AladjevProcedures`”}}
In[2662]:= G[x_] := ToString1[x]; CallsInProc[G]

Out[2662] = {{”Close“, “DeleteDuplicates“, “Flatten“, “For“, “If“, “MemberQ“,
“Read“, “Return“, “StringLength“, “StringQ“, “StringTake“, “StringTrim“, “While“,
“Write”}, {”StrDelEnds“, “SuffPref“, “ToString1”}, {”AladjevProcedures`”}}
```

The procedure call **CallsInProc[*x*]** returns the nested*3*–element list whose the first element defines the list of standard functions, the second element defines the list of external and internal functions, blocks and modules of the user, whose calls uses an object*x* whereas the third element defines the list of contexts which correspond to the user means and which are used by the object*x.* The**CallsInProc** procedure represents essential interest for analysis of the user means regarding existence of calls in them of both the user tools, and the system software.

For operating with procedures and functions, whose definitions have been evaluated in the current session, a simple**CsProcsFuncs** function is a rather useful means whose call**CsProcsFuncs[]** returns the list of blocks, functions and modules, whose definitions were evaluated in the current session. The fragment represents source code of the function with an example of its use.

```
In[2719] := CsProcsFuncs[] := Select[CNames[“Global`”], ProcQ[#] || FunctionQ[#]
&]
In[2720]:= CsProcsFuncs[]
Out[2720]= {”A“, “ArtKr“, “Av“, “B“, “H72“, “Kr“, “V“, “Vg“, “W”}
```
Naturally, the given list doesn’t include the procedures and functions from the packages uploaded into the current session, of both system means, and user means for the reason that similar means are associated with contexts of the corresponding packages. Moreover, due to the necessity of analysis of a quite large number of means of the current session the performance of this function can demand noticeable temporary expenses. The**CsProcsFuncs1** procedure is a rather useful modification of the previous function

whose call**CsProcsFuncs1[]** returns the nested list whose elements define lists whose*first* elements define means similarly to the**CsProcsFuncs** function while the*second* elements– multiplicities of their definitions. The following fragment represents source code of the**CsProcsFuncs1** procedure along with typical examples of its usage.

In[2532]:= **CsProcsFuncs1[] := Module[{a = CsProcsFuncs[], b, c}, b = Map[Definition2, ToExpression[a]]; c = Quiet[Mapp[Select, b, StringFreeQ[#1, ToString[#1] <> "Options[" <> ToString[#1] <> "] := "] &]]; Select[Map9[List, a, Map[Length, c]], ! MemberQ[#, "CsProcsFuncs1"] &]]** In[2533]:= **CsProcsFuncs1[]**
Out[2533]= {{"LocalVars", 1}, {"V", 4}, {"W", 2}, {"Z", 2}, {"Art", 6}, {"Kr", 4}}

Analogously to the **CsProcsFuncs** function, the call**CsProcsFuncs1[]** of the previous procedure because of necessity of analysis of a quite large number of the means which are activated in the current session can demand enough noticeable temporary expenses.
The next procedure**ActCsProcFunc** is a means rather useful in the practical relation, its call**ActCsProcFunc[]** returns the nested two–element list whose elements are sublists of variable length. The first element of the*first* sublist– ***"Procedure"*** while others define the***2***–element lists containing names of the procedures with their headings activated in the current session. Whereas the first element of the*second* sublist– ***"Function"*** whereas others determine the ***2***–element lists containing names of the functions with their headings which were activated in the current session. At that, the procedures can contain in own composition both the blocks, and the modules. The following fragment represents source code of the**ActCsProcFunc** procedure along with the most typical examples of its usage.

In[2742] **:= ActCsProcFunc[] := Module[{a = Names["Global`*"], h = {}, d, t, b = {"Procedure"}, c = {"Function"}, k = 1, v}, Map[If[TemporaryQ[#] || HeadPF[#] === #, Null, AppendTo[h, ToString[t = Unique["g"]]]; v = BlockFuncModQ[#, t]; If[v && MemberQ[{"Block", "Module"}, t], AppendTo[b, {#, HeadPF[#]}], If[v && t === "Function", AppendTo[c, {#, HeadPF[#]}]]]]] &, a]; Map[Remove, h]; {b, c}]**

In[2743] **:= TemporaryQ[x_] := If[SymbolQ[x], MemberQ[{"Attributes[" <> ToString[x] <> "] = {Temporary}", "Null"}, ToString[Definition[x]]], False]**
In[2744]**:= Map[TemporaryQ, {gs47, gs, a + b}]**

Out[2744] = {True, True, False}
In[2745]:= **g[x_] := Module[{}, x]; s[x_, y_] := Block[{}, x + y]; v[x_] := x; n[x_] := x; vs[x_, y_] := x + y; gs[x_] := x^2; hg[x___] := Length[{x}]; hh[x_, y_] := x^2 + y^2; nm[x_, y_] := Module[{}, x*y]; ts[x_Integer] := Block[{a = 72}, x + a]; w[x_] := x; w[x_, y_] := x*y;**
In[2746]:= **ActCsProcFunc[]**
Out[2746]= {{"Procedure",{"g", "g[x_]"},{"nm", "nm[x_,y_]"}, {"s", "s[x_,y_]"}, {"ts", "ts[x_Integer]"}},
{"Function", {"gs", "gs[x_]"},{"hg", "hg[x___]"},{"hh", "hh[x_,y_]"}, {"n", "n[x_]"}, {"TemporaryQ", {"TemporaryQ[x_/;SymbolQ[x]]", "TemporaryQ[x_]"}}, {"v", "v[x_]"}, {"vs", "vs[x_, y_]"}, {"w", {"w[x_]", "w[x_, y_]"}}}}
In[2747]:= **A[___] := Module[{a, b = 590}, Map[TemporaryQ, {a, b}]]; A[]**
Out[2747]= {True, False}

The given procedure materially uses the **TemporaryQ** function, whose call

TemporaryQ[*x*] returns*True* if a symbol***x*** defines the temporary variable, and*False* otherwise. In particular, for a local variable***x*** without initial value the call**TemporaryQ[*x*]** returns*True***.** The**TemporaryQ** function is useful in many appendices, above all of the system character. The previous fragment represents source code of the function with typical examples of its usage. Analogously to the means**CsProcsFuncs** and**CsProcsFuncs1** the procedure call**ActCsProcFunc** because of necessity of analysis of a quite large number of the means which are activated in the current session can demand enough noticeable temporary expenses. Concerning the**ActCsProcFunc** procedure it should be noted that it provides return only of blocks, functions, modules whose*definitions* have been evaluated in the***Input****–paragraph* mode without allowing to receive objects of this type which were loaded into the current session in the***Input****–*paragraph mode, in particular, as a result of uploading of the user package by means of the**LoadMyPackage** procedure as visually illustrate the last examples of the previous fragment. The reason for this is that this objects are associated with the*context* of a package containing them but not with the context**"Global`".**

As it was noted above, the strict differentiation of objects in environment of the**Mathematica** is carried out not by their*names***,** but by their*headings***.** For this reason in a number of cases of procedural programming the problem of organization of mechanisms of the differentiated processing of such objects on the basis of their headings arises. Certain such means is presented in the present book, here we will define*two* procedures ensuring the differentiated operating with attributes of such objects. Unlike the**Rename** procedure, the **RenameH** procedure provides in a certain degree selective renaming of the blocks, functions, modules of the same name on the basis of their headings. The successful call**RenameH[*x*,*y*]** returns*Null***,** i.e. nothing, renaming an object with a heading***x*** onto a name***y*** with saving of attributes**;** at that, the initial object with heading***x*** is removed from the current session.

In[2563]:= **RenameH[x_ /; HeadingQ1[x], y_ /; ! HowAct[y], z___] :=**

Module[{c, a = HeadName[x], d = StandHead[x], b = ToExpression["Attributes[" <> HeadName[x] <> "]"]}, c = Flatten[{PureDefinition[a]}]; If[c == {$Failed}, $Failed, If[c == {}, Return[$Failed], ToExpression["ClearAllAttributes[" <> a <> "]"]]; ToExpression[ToString[y] <> DelSuffPref[Select[c, SuffPref[#, d <> " := ", 1] &][[1]], a, 1]]; If[{z}== {}, RemProcOnHead[d]]; If[! SameQ[PureDefinition[a], $Failed], ToExpression["SetAttributes["<> ToString[a] <> "," <> ToString[b] <> "]"]]; ToExpression["SetAttributes[" <> ToString[y] <> "," <> ToString[b] <> "]"];]]
In[2564]:= **M[x_ /; SameQ[x, "avz"], y_] := Module[{a, b, c}, y]; M[x_, y_] := Module[{a, b, c}, "agn"; x + y]; M[x_String] := x;**

M[x_ /; ListQ[x], y_] := Block[{a, b, c}, "agn"; Length[x] + y]; SetAttributes[M, Protected] In[2565]:= **RenameH["M[x_,y_]", V]**
In[2566]:= **Definition[V]**
Out[2566]= Attributes[V]= {Protected}

V[x **_**, y_]**:=** Module[{a, b, c}, **"**agn**"; x**+ y]
In[2567]:= **Definition[M]**
Out[2567]= Attributes[M]= {Protected}

M[x _ /; x=== "avz", y_]:= Module[{a, b, c}, y]
M[x_ /; ListQ[x], y_]:= Block[{a, b, c}, "agn"; Length[x]+ y] M[x_String]:= x

In[2568] **:= RenameH["M[x_String]", S, 90]**
In[2569]**:= Definition[S]**
Out[2569]= Attributes[S]= {Protected}

S[x _String]:= x
In[2570]**:= Definition[M]**
Out[2570]= Attributes[M]= {Protected}

M[x _ /; x=== "avz", y_]:= Module[{a, b, c}, y]
M[x_ /; ListQ[x], y_]:= Block[{a, b, c}, "agn"; Length[x]+ y] M[x_String]:= x

At that, the procedure call **RenameH[*x, y, z*]** with the*3rd* optional argument ***z*** *–an arbitrary expression*– renames an object with heading***x*** onto a name***y*** with saving of the attributes; meanwhile, the object with heading***x*** remains active in the current session. On an inadmissible tuple of factual arguments the procedure call returns**$Failed** or returned unevaluated. The previous fragment represents source code of the**RenameH** procedure along with the most typical examples of its usage.

In a number of the procedures intended for processing of definitions or calls of other procedures/functions, the problem of*identification* of the call format, i.e. format of type***F*[*args*]** where***F*** – the name of a procedure/function and ***args*** – the tuple of formal or factual arguments is a rather topical. The next fragment represents source code of the**CallQ** procedure along with typical enough examples of its usage.

In[2540] **:= CallQ[x_] := Module[{b, c, a=ToString[If[Quiet[Part[x, 1]] ===–1, Part[x, 1]*x, x]]}, b = Flatten[StringPosition[a, "["]]; If[b == {}, False, c = b[[1]]; If[SymbolQ[StringTake[a, {1, c–1}]] && StringTake[a, {c + 1, c + 1}] != "[" && StringTake[a,–1] == "]", True, False]]]** In[2541]:= **CallQ[A[x, y, z]]**

Out[2541] = True
In[2542]**:= Map[CallQ, {Sin[–90], Sin[9.0]}]**
Out[2542]= {True, False}

In[2543] **:= FormalArgs[x_] := Module[{a, b = Quiet[Part[x, 1]]}, If[CallQ[x], a = ToString[If[b ===–1, Part[x, 1]*x, x]]; ToExpression["{" <> StringTake[a, {Flatten[StringPosition[a, "["]][[1]] + 1,–2}] <> "}"], $Failed]]**

In[2544] **:= Map[FormalArgs, {Agn[x, y, x], Sin[–a + b], Agn[x_ /; StringQ[x], y_Integer, z_]}]**
Out[2544]= {{x, y, x}, {a– b}, {x_ /; StringQ[x], y_Integer, z_}}
In[2545]**:= Map[FormalArgs, {Agn[], a + b, 90, {a, b, c}}]**
Out[2545]= {{}, $Failed, $Failed, $Failed}

The procedure call **CallQ[*x*]** up to a sign returns*True* if***x*** is an expression of the format***F*[*args*]** where***F*** – name of a procedure/function and***args*** – tuple of the actual arguments, and*False* otherwise. The above**CallQ** procedure is of interest as a testing means for checking of the actual arguments of objects for their admissibility. While the procedure call**FormalArgs[*x*]** returns the list of formal arguments of a heading***x*** irrespectively off definition ascribed to it; on an*inadmissible* heading***x*****$Failed** is returned.

The previous fragment represents source code of the procedure along with an example of its usage.

In the **Maple** system in problems of procedural programming the*procedural* ***"procname"*** variable is rather useful, whose use in the body of a procedure allows to receive the heading of procedure in a point of its call. The variable is useful enough at realization of some special mechanisms of processing in procedures what was rather widely used by us for programming of system means expanding the software of the**Maple** system [10-22,25-27,47]. Similar means in the**Mathematica** system are absent, meanwhile, means of similar character are useful enough at realization of the procedural paradigm of the system. As one useful means of this type it is quite possible to consider the **$InBlockMod** variable whose call in the body of a block or module in string format returns source code of an object containing it without a heading in a point of its call. The next fragment adduces source code of the**$InBlockMod** variable along with examples of its typical usage.

In[2550]:= **StringReplace3[S_/; StringQ[S], x__] := Module[{b = S, c, j = 1, a = Map[ToString, {x}]}, c = Length[a]; If[OddQ[c], S, While[j <= c/2, b = StringReplace2[b, a[[2*j–1]], a[[2*j]]]; j++]; b]]**

In[2551] **:= StringReplace3["Module[{a$ = 78, b$ = 90, c$ =72}, xb$; a$*b$*6; (a$+b$+c$)*(x+y); aa$]", "a$", "a", "b$", "b", "c$", "c"]**
Out[2551]= "Module[{a= 78, b= 90, c= 72}, xb$; a*b*6; (a+b+c)*(x+y); aa$]"

In[2552] **:= $InBlockMod := Quiet[Check[StringTake[If[Stack[Block] != {}, ToString[InputForm[Stack[Block][[1]]]], If[Stack[Module] != {}, StringReplace3[ToString[InputForm[Stack[Module][[1]]]], Sequences[Riffle[Select[StringReplace[StringSplit[StringTake[SubStrSymbolParity1[ToString[InputForm[Stack[Module][[1]]]], "{", "}"][[1]], {2,–2}], " "], ","–> ""], StringTake[#,–1] == "$" &], Mapp[StringTake, Select[StringReplace[StringSplit[StringTake[SubStrSymbolParity1[ToString[InputForm[Stack[Module][[1]]]], "{", "}"][[1]], {2,–2}], " "], ","–> ""], StringTake[#,–1] == "$" &], {1,–2}]]]]], $Failed], {10,–2}], Null]]**

In[2553]:= **Avz[x_] := Block[{a = 6, b =50, c =$InBlockMod}, Print[c]; a*b*x]**
In[2554]:= **Avz[42]**

" Block[{a= 6, b= 50, c= $InBlockMod}, Print[c]; a*b*42]**"** Out[2554]= 12 600
In[2555]:= **Agn[x_] := Module[{a=6, b=50, c=$InBlockMod}, Print[c]; a*b*x]**
In[2556]:= **Agn[47]**

"Module[{a= 6, b= 50, c= $InBlockMod}, Print[c]; a*b*47]**"** Out[2556]= 14 100
In[2557]:= **Avs[x_] := Module[{a = $InBlockMod, b = 50, c = 500}, Print[a]; b*c*x^2]: Avs[500]**

" Block[{a= $InBlockMod, b= 50, c= 500}, Print[a]; b*c*500^2] Out[2557]= 6 250 000 000
In[2558]:= **Av[x_] := Module[{a = $InBlockMod, b = 50, c = 500}, Print[a];**

b*c*x^2]: Av[660] "Module[{a= $InBlockMod, b= 50, c= 500}, Print[a]; b*c*660^2]**"**
Out[2558]= 10 890 000 000

In[2559]:= **$InBlockMod**
In[2560]:= **G[x_, y_] := {x + y, $InBlockMod}**; **G[42, 47]**
Out[2560]= {89, Null}

At that, for realization of algorithm of the above variable the**StringReplace3** procedure which is an expansion of the**StringReplace2** procedure is rather significantly used. Its source code with examples of application is presented in the beginning of the previous fragment. The call**StringReplace3[*W,x, x1, y, y1, z, z1, …*]** returns the result of substitution into a string***W*** of substrings {***x1, y1, z1,…***} instead of all occurrences of substrings {***x, y, z,…***} accordingly; in the absence of such occurrences the call returns an initial string***W.*** This procedure appears as a very useful tool of processing of string constructions which contain expressions, expanding possibilities of the standard means. At using of the procedural variable**$InBlockMod** it must be kept in mind that it makes sense only in the body of a procedure of type {***Block, Module***}, returning nothing, i.e.*Null***,** in other expressions or in an***Input***–paragraph as visually illustrate examples of application of the variable**$InBlockMod** in the previous fragment. At that it must be kept in mind in order to avoid of misunderstanding the call of the variable**$InBlockMod** is recommended to do at the beginning of procedures, for example, in area of local variables.

The next procedure is useful for operating with blocks/functions/modules. The procedure call**FullUserTools[*x*]** returns the list of names, that enter in definition of the active user block/function/module***x;*** in addition, the first element of the list is a context of these tools. Whereas in a case of tools with various contexts a call returns the nested list of sublists of the above format. In turn, a procedure call**FullUserTools[*x, y*]** thru the optional argument***y*** **–** *an undefinite variable***–** returns***2***–element list whose the*first* element defines list of tools without*usages***,** and the*second* element defines*unidentified* tools. The next fragment represents source code of the procedure with examples.

In[2718]:= **FullUserTools[x_ /; BlockFuncModQ[x], y___] := Module[{a, b, c, d, p = {}, n = {}}, Save[Set[a, ToString[x] <> ".txt"], x]; b = ReadString[a]; DeleteFile[a]; c = StringSplit[b, "\r\n \r\n"];**

b = Select[c, ! StringFreeQ[#, "::usage = ""] &]; d = MinusList[c, b]; c = Map[StringSplit[#, " /: ", 2][[1]] &, b]; d = Map[StringSplit[#, " := "][[1]] &, d]; Quiet[Map[If[HeadingQ[#], AppendTo[p, HeadName[#]], AppendTo[n, #]] &, d]]; {a, p}= {Join[c, p], MinusList[c, p]}; b = Map[MinusList[#, {ToString[x]}] &, {a, p}][[1]]; b = DeleteDuplicates[Map[{#, Context[#]}&, b]]; b = Gather[b, #1[[2]] == #2[[2]] &]; b = Map[Sort[DeleteDuplicates[Flatten[#]]] &, b]; d = Map[Sort[#, ContextQ[#1] &] &, b]; d = Map[Flatten[{#[[1]], Sort[#[[2 ;;–1]]]}] &, d]; d = If[Length[d] == 1, d[[1]], d]; If[{y}!= {}&& ! HowAct[y], y = {p, n}; d, d]]

In[2719]:= **FullUserTools[UnevaluatedQ]**
Out[2719]= {"AladjevProcedures`", "ListStrToStr", "StrDelEnds", "SuffPref", "SymbolQ", "ToString1"}
In[2720]:= **F[x_, y_] := Module[{a = 90, b = 500, c}, a*b*x*y; c = ToString1[c]];**

Sv[x_, y_] := x*y; G[x_, y_] := Module[{}, {ToString1[x*y], F[x] + Sv[x, y]}]
In[2721]:= **FullUserTools[G]**
Out[2721]= {{"AladjevProcedures`", "StrDelEnds", "SuffPref", "ToString1"},

{"Global`", "F", "Sv"}}

Unlike the**FullUserTools** procedure the**FullToolsCalls** procedure provides the analysis of the user block, function or module regarding existence in its definition of calls of both the user and the system means. The procedure call **FullToolsCalls[*x*]** returns the list of names, whose calls are in definition of the active user block/function/module***x;*** in addition, the first element of the list is a context of these tools. While in case of means with different contexts the procedure call returns the nested list of sublists of the above format. In case of absence in a***x*** definition of the user or system calls the procedure call **FullToolsCalls[*x*]** returns the empty list, i.e. {}.

In[2920] **:= FullToolsCalls[x_ /; BlockFuncModQ[x]] := Module[{b, c = {}, d, a = Flatten[{PureDefinition[x]}][[1]], k = 1, g = {}, p, j, n}, b = Gather[Map[#[[1]] &, StringPosition[a, "["]], Abs[#1–#2] == 1 &]; b = Flatten[Select[b, Length[#] == 1 &]]; For[k, k <= Length[b], k++, n = ""; For[j = b[[k]]–1, j >= 0, j—, If[SymbolQ[p = Quiet[StringTake[a, {j}]]] || IntegerQ[Quiet[ToExpression[p]]], n = p <> n, AppendTo[c, n]; Break[]]]]; c = MinusList[c, Join[Locals[x], Args[x, 90], {"Block", "Module"}]]; c = Map[{#, Quiet[Context[#]]}&, MinusList[c, {ToString[x]}]];**

b = Map[Sort[DeleteDuplicates[Flatten[#]]] &, c]; d = Map[Flatten, Gather[Map[Sort[#, Quiet[ContextQ[#1]] &] &, b], #1[[1]] == #2[[1]] &]]; d = Map[Flatten[{#[[1]], Sort[#[[2 ;;–1]]]}] &, Map[DeleteDuplicates, d]]; d = If[Length[d] == 1, d[[1]], d]; Select[d, ! Quiet[SameQ[#[[1]], Context[""]]] &]]

In[2921] **:= AH[x_] := (Sv[x] + GSV[x, 90, 500])*Sin[x] + Z[[a]] + Art[x]/Kr[x]**
In[2922]**:= FullToolsCalls[AH]**
Out[2922]= {{"Tallinn`", "Sv"}, {"RansIan`", "GSV"}, {"System`", "Sin"},

{ "Global`", "Art", "Kr"}}
In[2923]**:= FullToolsCalls[UnevaluatedQ]**
Out[2923]= {{"AladjevProcedures`", "ListStrToStr", "SymbolQ"},

{"System`", "Check", "If", "Module", "Quiet", "StringJoin", "ToString"}} Unlike the previous procedure the**FullToolsCallsM** procedure provides the above analysis of the user block, function or module of the same name.

In[2954] **:= FullToolsCallsM[x_ /; BlockFuncModQ[x]] := Module[{b, c = {}, a = Flatten[{PureDefinition[x]}], k = 1, n = ToString[x]}, If[Length[a] == 1, FullToolsCalls[x], For[k, k <= Length[a], k++, b = ToString[Unique["sv"]]; ToExpression[StringReplace[a[[k]], n <> "["–> b <> "[", 1]]; AppendTo[c, FullToolsCalls[b]]; Quiet[Remove[b]]]; c = Map[If[NestListQ[#] && Length[#] == 1, #[[1]], #] &, c]; Map[If[! NestListQ[#] && Length[#] == 1, {}, If[! NestListQ[#] && Length[#] >1, #, Select[#, Length[#]> 1 &]]] &, c]]]**

In[2955] **:= Ah[x_] := (Sv[x]+ GSV[x, 90, 500])*Sin[x] + Z[[a]] + Art[x]/Kr[x]; Ah[x_Integer] := Block[{a = 90}, ToString1[a*Cos[x]]]; Ah[x_String] := Module[{a ="6"}, ToString1[x <>a]], FullToolsCallsM[Ah]** Out[2956]= {{"System`", "Cos"}, {"System`", "StringJoin"}, {{"Tallinn`", "Sv"},

{ "RansIan`", "GSV"}, {"System`", "Sin"}, {"Global`", "Art", "Kr"}}} In[2957]**:= G[x_, y_, z_] := Module[{}, x*y*z]; G[x_] := Module[{a = 6}, x/a]** In[2958]**:=**

FullToolsCallsM[G]
Out[2958]= {{}, {}}
In[2958]:= **ProcQ[x_, y_] := Block[{a=0, b=1}, ToString1[a*Sin[x]+b*Cos[y]]]**
In[2959]:= **FullToolsCallsM[ToString1]**
Out[2959]= {{"System`", "Close", "DeleteFile", "For", "If", "Read", "Return",

" StringJoin", "Write"}, {"AladjevProcedures`", "StrDelEnds"}} In[2960]:= **Avz[x_] := Module[{a = 90}, StrStr[x] <> ToString[a]]** In[2961]:= **Map[#[Avz] &, {FullToolsCalls, FullToolsCallsM}]** Out[2961]= {{{"System`", "StringJoin", "ToString"}, {"AladjevProcedures`",

"StrStr"}}, {{"System`", "StringJoin", "ToString"}, {"AladjevProcedures`", "StrStr"}}}

The procedure call **FullToolsCallsM[*x*]** returns the nested list of results of application of the**FullToolsCalls** procedure to subobjects*(blocks*,*functions*, *modules)* that compose an object of the same name***x*.** The order of elements in the returned list corresponds to an order of definitions of the subobjects returned by the call**Definition[*x*].** Whereas, the procedure call**AllCalls[*x*]** returns the nested list of sublists containing the full form of calls entering in definitions of*subobjects* that compose an object of the same name or a*simple* object***x*.** The order of elements in the returned list corresponds to an order of definitions of the subobjects returned by the call**Definition[*x*].**

In[2840] **:= AllCalls[x_ /; BlockFuncModQ[x]] := Module[{a1, ArtKr, k1, b1, c1 = {}, d1, m = ToString[x]}, a1 = Flatten[{PureDefinition[x]}]; ArtKr[y_] := Module[{a = Flatten[{PureDefinition[y]}][[1]], b, c = {}, d, k, g = {}, p, j, n}, b = Gather[Map[# [[1]] &, StringPosition[a, "["]], Abs[#1–#2] == 1 &]; b = Flatten[Select[b, Length[#] == 1 &]]; For[k = 1, k <= Length[b], k++, n = ""; For[j = b[[k]]–1, j >= 0, j—, If[SymbolQ[p = Quiet[StringTake[a, {j}]]] || IntegerQ[Quiet[ToExpression[p]]], n = p <> n, AppendTo[c, n]; Break[]]]]; For[k = 1, k <= Length[b], k++, For[j = b[[k]], j <= StringLength[a], j++, SubStrSymbolParity1[StringTake[a, {j, StringLength[a]}], "[", "]"][[1]]; AppendTo[g, SubStrSymbolParity1[StringTake[a, {j, StringLength[a]}], "[", "]"][[1]]]; Break[]]]; n = Select[Map[StringJoin[#] &, Partition[Riffle[c, g], 2]], # ! = HeadPF[y] &]; If[FunctionQ[y], n, n[[2 ;;–1]]]]]; If[Length[a1] == 1, ArtKr[x], For[k1 = 1, k1 <= Length[a1], k1++, b1 = ToString[Unique["v"]]; ToExpression[StringReplace[a1[[k1]], m <> "["–> b1 <> "[", 1]]; AppendTo[c1, ArtKr[b1]]; Quiet[Remove[b1]]]; c1]]**

In[2841] **:= AH[x_] := (Sv[x] +GSV[x, 90, 500])*Sin[x]+ Z[[a]] + Art[x]/Kr[x]; AH[x_Integer] := Block[{a = 90}, ToString1[a*Cos[x]]]; AH[x_String] := Module[{a="500"}, ToString1[x <>a]]; AH1[x_] := (Sv[x]+GSV[x,90,500])* Sin[x] + Z[[a]] + Art[x]/Kr[x]; F[x_, y_] := x + y**

In[2842]:= **AllCalls[AH]**
Out[2842]= {{"ToString1[a*Cos[x]]", "Cos[x]"}, {"ToString1[StringJoin[x, a]]",

" StringJoin[x, a]"}, {"Sv[x]", "GSV[x, 90, 500]", "Sin[x]", "Art[x]", "Kr[x]"}}
In[2843]:= **AllCalls[F]**
Out[2843]= {}
In[2844]:= **AllCalls[AH1]**
Out[2844]= {"Sv[x]", "GSV[x, 90, 500]", "Sin[x]", "Art[x]", "Kr[x]"}

On that the presentation of tools, serving for processing of the user objects, is completed**;** at that, some tools accompanying them are considered below or were already considered above. Classification of our tools has in a certain measure a subjective character that is caused by their basic use or frequency of usage at programming of the means represented in the given book and in a number of important applications of the applied and the system character. These means are mainly used at programming of the system tools.

Chapter 7. Means of input–output of the*Mathematica*

The **Mathematica** language being the built–in programming language that first of all is oriented onto symbolical calculations and processing has rather limited facilities for data processing which first of all are located in external memory of the computer. In this regard the language significantly concedes to the traditional programming languages***C++, Basic, Fortran, Cobol, PL/1, ADA, Pascal,*** etc. At the same time, being oriented, first of all, onto solution of tasks in symbolic view, the**Mathematica** language provides a set of tools for access to datafiles that can quite satisfy a rather wide range of the users of mathematical applications of the**Mathematica**. In this chapter the means of access to datafiles are considered rather superficially owing to the limited volume, extensiveness of this theme and purpose of the present book. The reader who is interested in means of access to datafiles of the**Mathematica** system quite can appeal to documentation delivered with the system. At the same time, for the purpose of development of methods of access to file system of the computer we created a number of rather effective means that are represented in the***AVZ_Package*** package [48]. Whereas in the present chapter the attention is oriented on the means expanding standard means of the **Mathematica** system for ensuring work with files of the computer. Some of them are rather useful to practical application in the environment of the**Mathematica** system.

7.1. Means of the*Mathematica*for work with internal files

Means of ***Math***language provide access of the user to files of several types which can be conditionally divided into two large groups, namely***:****internal* and*external* files. During the routine work the system deals with***3*** various types of internal files from which we will note the files having extensions {***“nd”, “m”, “mx”***}, their structure is distinguished by the standard system means and which are important enough already on the first stages of work with system. Before further consideration we will note that the concept of the*file qualifier* ***(FQ)*** defining the full path to the required file in file system of the computer or to its subdirectory,practically, completely coincides with similar concept for already mentioned**Maple** system excepting that if in the **Maple** for***FQ*** the format of type {***string, symbol***} is allowed whereas for***FQ*** in the**Mathematica** system the***string–***type format is admissible only.

The call **Directory[]** of the system function returns an active subdirectory of the current session of the system whereas the call**SetDirectory[*x*]** returns a directory***x,*** doing it active in the current session**;** at that, as an*active (current)* directory is understood the directory whose files are processed by means of access if only their names, but not full paths to them are specified. At that, defining at the call**SetDirectory[*x*]** the system**$UserDocumentsDirectory** variable as a factual***x***–argument, it is possible to redefine the user current subdirectory by default. Meanwhile, the**SetDirectory** function allows only real–life subdirectories as the argument, causing on nonexistent directories the erroneous situation with returning**$Failed.** On the other hand, a rather simple**SetDir** procedure provides possibility to determine also nonexistent subdirectories as the current subdirectories. The procedure call**SetDir[*x*]** on an existing subdirectory***x*** does it current while a nonexistent subdirectory is previously created and then it is defined by the current subdirectory. At that, if the factual***x***argument at the call**SetDir[*x*]** is determined by a chain without name of the***IO*** device, for example,**"**aa**…**\bb**",** then a chain of the subdirectories**Directory[] <> "**aa**…**\bb**"** is created that determines a full path to the created current subdirectory. The next fragment represents source code of the**SetDir** procedure along with the most typical examples of its usage.

In[2531] **:= Directory[]**
Out[2531]= **"**C**:**\Users\Aladjev\Documents**"**
In[2532]**:= SetDirectory["E:\AVZ_Package"]**

SetDirectory **::**cdir**:** Cannot set current directory to E**:**\AVZ_Package**.** >> Out[2532]= **$**Failed
In[2533]**:= SetDirectory[$UserDocumentsDirectory]**
Out[2533]= **"**C**:**\Users\Aladjev\Documents**"**
In[2534]**:= SetDirectory[]**
Out[2534]= **"**C**:**\Users\Aladjev**"**

In[2535] **:= SetDir[x_/; StringQ[x]] := Module[**{**a**}**, If[StringLength[x] == 1 || StringLength[x] >= 2 && StringTake[x,** {**2, 2**}**] != ":", Return[Quiet[SetDirectory[Quiet[CreateDirectory[**

StringReplace[Directory[] <> "\" <> x, "\\" –> "\"]]]]]], Null]; a = Quiet[CreateDirectory[StringTake[x, 1] <> ":\"]]; If[a === $Failed, Return[$Failed], Null]; Quiet[Check[If[DirectoryQ[x], SetDirectory[x], SetDirectory[CreateDirectory[x]]], Null]]; Directory[]]

In[2536] **:= SetDir["C:\Temp\111\222\333\444\555\666\777"]** Out[2536]= **"**C**:**\Temp\111\222\333\444\555\666\777**"** In[2537]**:= SetDir["H:\111\222\333\444\555\666\777\888"]** Out[2537]= **$**Failed
In[2538]**:= Directory[]**
Out[2538]= **"**C**:**\Temp\111\222\333\444\555\666\777**"** In[2539]**:= SetDir["kr\6"]**
Out[2539]= **"**C**:**\Temp\111\222\333\444\555\666\777\kr\6**"** In[2540]**:= SetDir["E:\AVZ_Package"]**
Out[2540]= **"**E**:**\AVZ_Package**"**

In[2541] **:= Adrive[] := Module[**{**a, b, c, d, k = 1**}**,**
{**a, b**}**=** {**CharacterRange["A", "Z"],** {}}**; For[k, k <= 26, k++, c = a[[k]] <> ":\";**

d = Quiet[CreateDirectory[c]]; If[d === $Failed, Null, AppendTo[b, StringTake[d, 1]]]]; Sort[b]] In[2542]:= **Adrive[]**
Out[2542]= {**"**C**"**, **"**D**"**, **"**E**"**, **"**F**"**, **"**G**"**}

Meanwhile, in attempt of definition of a nonexistent directory as the current directory the emergence of a situation is quite real when as a***IO*** device has been specified a device

which at the moment isn't existing in the system or inaccessible. Therefore rather actually to have the means allowing to verify availability of*IO* devices in the system. In this regard the**Adrive** procedure solves this problem, whose call**Adrive[]** returns the list of logical names in string format of*IO* devices, available at the moment. The given procedure is an analog of the procedure of the same name for the**Maple** system [47], the last part of the previous fragment represents source code of the**Adrive** procedure with an example of its usage. Both procedures of the previous fragment are useful enough at programming in the**Mathematica** system of various means of access to the datafiles.
The following**Adrive1** procedure expands the above**Adrive** procedure and returns the***returns the***element nested list whose first element represents the list with names in string format of all active direct access devices whereas the second element represents the list with names in string format of all inactive direct access devices of the computer. The next fragment represents source code of the**Adrive1** procedure along with a typical example of its usage.

In[2560] **:= Adrive1[] := Module[{a = CharacterRange["A", "Z"], b = {}, c = 1, d, p = {}, h, t = "$Art26$Kr18$"}, For[c, c <= 26, c++, d = a[[c]] <> ":\"; If[DirQ[d], AppendTo[b, a[[c]]]; h = Quiet[CreateDirectory[d <> t]]; If[h === $Failed, Continue[], DeleteDirectory[d <> t]; AppendTo[p, a[[c]]]; Continue[]]]]; {p, MinusList[b, p]}]**

In[2561]**:= Adrive1[]**
Out[2561]= {{"C", "D", "G"}, {"A", "E"}}
In[2562]**:= SetDir1[x_ /; StringQ[x]] := Module[{a = SetDir[x], b, c, k},**

If[! SameQ[a, $Failed], a, k = 1; b = Adrive[]; c = Map[FreeSpaceVol, b]; While[k <= Length[b], PrependTo[c[[k]], b[[k]]]; k++]; c = SortNL1[c, 2, Greater]; SetDir[StringJoin[c[[1]][[1]], StringTake[x, {2,–1}]]]]]]

In[2563]**:= SetDir1["G:\Galina/Svetla\ArtKr/Tampere\Tallinn"]** Out[2563]= "C:\Galina/Svetla\ArtKr/Tampere\Tallinn"

At last, the**SetDir1** procedure presented at the end of the previous fragment expands the**SetDir** procedure onto the case when attempt to create a chain of directories meets the especial situation caused by lack of the demanded device on which creation of this chain of subdirectories was planned. In the absence of such device of direct access the procedure call**SetDir1[*x*]** returns the created chain of subdirectories on a device having the greatest possible volume of available memory among all active devices of direct access in the current session of the**Mathematica** system.

Files with documents which in one of ***11*** formats by the chain of command ***"File –> {Save As|Save}"*** of the***GUI****(the most used formats****"nb", "m")*** are saved, the files with the***Mathematica–***objects saved by the**Save** function *(input format)***,** and datafiles with**Mathematica** packages*(format****"m", "mx")*** belong to the internal files. These files represent quite certain interest at the solution of many problems demanding both the standard methods, and the advanced methods of programming. For standard support of operating with them the**Mathematica** system has a number of means whereas for ensuring expanded work with similar datafiles a set of means can be created, some of which are considered in the present book and also have been included to our ***AVZ_Package*** package

[48]. At that, files of any of the*specified* formats with the definitions of objects saved in them by the**Save** function as a result of uploading of these files by the**Get** function into the subsequent sessions of the system provide availability of these objects.

It is rather simple to be convinced that the datafiles created by means of the **Save** function contain definitions of objects in*Input* ***Mathematica*–**format irrespective of extension of a datafile name. It provides possibility of rather simple organization of processing of such datafiles for various appendices. In particular, on the basis of structure of such datafiles it is possible without their uploading into the current session to obtain lists of names of the objects which are in them. For this purpose the**Nobj** procedure can be used, whose call**Nobj[*x,y*]** returns the list of names of the objects in string format which have been earlier saved in a datafile***x*** by means of the**Save** function while through the*second* actual argument***y*** the list of headings in string format of these objects is returned. Such decision is rather essential since in a datafile can be objects of the same name with various headings, exactly that identify uniqueness of an object.

At that, can arise a need not to upload by means of the **Get** function into the current session completely a file which has been earlier created by means of the**Save** function with activation of all objects containing in it, but to upload the objects containing in the file selectively, i.e. to create a kind of libraries of the user means. Concerning the packages created by means of a chain***"File →Save As→Mathematica Package (*.m)"*** of the***GUI*** commands, the given problem can be solved by means of the**Aobj** procedure, whose call**Aobj[*x, y*]** makes active in the current session all objects with a name***y*** from***m*–**file ***x*** which has been earlier created by the above chain of the***GUI*** commands. The fragment below represents source codes of the above procedures**Aobj** and**Nobj** along with the most typical examples of their usage.

In[2626]**:= Art1 := #^2 &; Art2 = #^3 &; Art3 := #^4 &; Art4 = #^5 &; Art := 26; Kr = 18; Agn[y_] := 67; Avz[x_] := 90*x + 500;**

SetAttributes[Avz, {Listable, Protected}] In[2627]**:= Save["C:/Temp/Obj.m", {Adrive, SetDir, Art1, Art2, Art3, Art4, Art, Nobj, Kr, Agn, Avz}]**

In[2628] **:= Nobj[x_ /; FileExistsQ[x] && StringTake[x,–2] == ".m", y_ /; ! HowAct[y]] := Module[{a, b, c, d, p, h, t, k = 1}, If[FileExistsQ[x] && MemberQ[{"Table", "Package"},**

Quiet[FileFormat[x]]], {a, b, d, h}= {OpenRead[x], {}, "90", {}}; While[! SameQ[d, "EndOfFile"], d = ToString[Read[a, String]]; If[! SuffPref[d, " ", 1], If[! StringFreeQ[d, "::usage = ""], AppendTo[b, StringSplit[StringTake[d, {1, Flatten[StringPosition[d, "::usage"]][[1]]–1}], " /: "][[1]]], p = Quiet[Check[StringTake[d, {1, Flatten[StringPosition[d, {" := ", " = "}]][[1]]–1}], $Failed]]; If[! SameQ[p, $Failed], If[SymbolQ[p]&& StringFreeQ[p, {" ", "{", "`"}]|| StringFreeQ[p, {" ", "{", "`"}] && HeadingQ1[p] === True, AppendTo[b, p]]]]]; k++]; Close[a]; b = Sort[DeleteDuplicates[b]]; h = Select[b, ! SymbolQ[#] &]; t = Map[If[SymbolQ[#],#, HeadName[#]] &, h]; b = MinusList[b, h]; b = Sort[DeleteDuplicates[Join[b, t]]]; y = MinusList[Sort[DeleteDuplicates[Join[h, Select[Map[If[! UnevaluatedQ[HeadPF, #], HeadPF[#]] &, b], ! SameQ[#, Null] &]]]], b]; b, $Failed]]

```
In[2629] := Clear[ArtKr]; Nobj["C:\Temp\Obj.m", ArtKr] Out[2629]= {"Adrive",
"Agn", "Art", "Art1", "Art2", "Art3", "Art4", "Avz", "BlockFuncModQ",
"ClearAllAttributes", "Contexts1", "Definition2", "HeadingQ", "HeadingQ1",
"HeadName", "HeadPF", "HowAct", "Kr", "ListStrToStr", "Map3", "Mapp",
"MinusList", "Nobj", "ProcQ", "PureDefinition", "RedSymbStr", "Sequences",
"SetDir", "StrDelEnds", "StringMultiple", "StringSplit1", "SubsDel", "SuffPref",
"SymbolQ", "SymbolQ1", "SysFuncQ", "SystemQ", "ToString1", "UnevaluatedQ"}
In[2630]:= ArtKr
Out[2630]= {"Adrive[]", "Agn[y_]", "Avz[x_]", "BlockFuncModQ[x_, y___]",
"ClearAllAttributes[x__]", "Contexts1[]", "Definition2[x_ /; SymbolQ[x]===
HowAct[x]]", "HeadingQ1[x_/; StringQ[x]]", "HeadingQ[x_/; StringQ[x]]",
"HeadName[x_ /; HeadingQ[x]|| HeadingQ1[x]]", "HowAct[x_]",
====================================================
"ToString1[x_]", "UnevaluatedQ[F_ /; SymbolQ[F], x___]"}

In[2650]:= Aobj[x_ /; FileExistsQ[x] && StringTake[x,–2] == ".m", y_ /; SymbolQ[y]
|| ListQ[y] &&

DeleteDuplicates[Map[SymbolQ[#] &, y]] == {True}] := Module[{a, b = "(*", c =
"*)", d = $AobjNobj, p = {Read[x, String], Close[x]}[[1]], h = Mapp[StringJoin,
Map[ToString, Flatten[{y}]], "["], k, j, g, s, t = {}, v = {}}, If[p != "(* ::Package:: *)",
$Failed, a = ReadFullFile[x]; If[StringFreeQ[a, d], $Failed, a = StringSplit[a, d][[2
;;–1]]; a = Map[StringReplace[#, {b–> "", c–> ""}] &, a]; a = Select[a, SuffPref[#, h,
1] &]; For[k = 1, k <= Length[h], k++, g = h[[k]]; For[j = 1, j <= Length[a], j++, s =
a[[j]];

c = StrSymbParity[s, g, "[", "]"]; c = If[c == {}, False,
HeadingQ1[Quiet[ToString[ToExpression[c[[1]]]]]] || HeadingQ[c[[1]]]];
If[SuffPref[s, g, 1] && c, AppendTo[t, s]; AppendTo[v, StringTake[g, {1,–2}]]]]]];
Map[ToExpression, t]; If[v !={}, Print["Software for " <> ToString[v] <> " is
downloaded"], Print["Software for " <> ToString[Flatten[{y}]] <> " was not
found"]]]]]

In[2651] := Art1[] := #^2 &
In[2652]:= Art2[] = #^3 &;
In[2653]:= Art3[] := #^4 &
In[2654]:= Art4[] = #^5 &;
In[2655]:= Art[] := 26
In[2656]:= Kr[] = 18;
In[2657]:= Agn[y_] := 67
In[2658]:= Avz[x_] := 90*x + 500
In[2659]:=Aobj["c:/tmp/Obj.m", {Art1, Art2, Art3, Art4, Art, Kr, Agn, Avz}]

Software for {Nobj , Avz, Agn, ArtKr, Sv} is downloaded
In[2659]:=Aobj["C:/Tmp/Obj.m", {Nobj90, Avz500, Agn67, Vsv47}]
Software for {Nobj90, Avz500, Agn67, Vsv47} was not found In[2660]:=
Map[PureDefinition, {Art1, Art2, Art3, Art4, Art, Kr, Agn, Avz}] Out[2660]=
{"Art1[]:= #1^2& ", "Art2[]= #1^3& ", "Art3[]:= #1^4& ", "Art4[]= #1^5& ",
"Art[]:= 26", "Kr[]= 18", "Agn[y_]:= 67", "Avz[x_]:= 90*x+ 500"}
```

The top part of the previous fragment represents a saving in a ***m***–file of the **Mathematica**–objects from this fragment and the objects given a little above in the same section. Further the source code of the**Nobj** procedure and an example of its application is represented. Right there it should be noted that *performance* of the**Nobj** procedure will demand certain temporary expenses. At that, if the main result of the procedure call**Nobj[*x, y*]** contains the list of names in string format of the means contained in a file***x,*** thru the second***y*** argument the headings of the means possessing them are returned.

Whereas the second part of the fragment represents source code of the **Aobj** procedure with an example of its use for activization in the current session of the objects {***Art1, Art2, Art3, Art4, Art, Kr, Agn, Avz***} which are in a***m***–file which is earlier created by means of chain***"File→Save As→Mathematica Package (*.m)"*** of the***GUI*** commands. Verification confirms availability of the specified objects in the current session. Moreover, as the*2nd* argument***y*** at the procedure call**Aobj** the separate symbol or their list can be. Besides that is supposed, before saving in a***m***–datafile***x*** all definitions of objects in the current document should have headings and be evaluated in separate ***Input***–paragraphs. The successful call**Aobj[*x, y*]** returns*Null,* i.e. nothing with output of the message concerning those means which were uploaded from a***m***-datafile***x*** or which are absent in the datafile. The procedures**Nobj** and**Aobj** process the main erroneous situations with returning on them the value**$Failed.** Both procedures can be extended by means of replenishment their by new useful enough functions.
The following**Aobj1** procedure is a rather useful extension of the previous **Aobj** procedure. Like the**Aobj** procedure the**Aobj1** procedure also is used for activation in the current session of the objects which are in a***m***–datafile which is earlier created by means of chain***"File→Save As→Mathematica Package(*.m)"*** of the*GUI* commands. The successful call**Aobj1[*x, y*]** returns *Null,* i.e. nothing with output of the messages concerning those means that were uploaded from a***m***file***x*** and that are absent in the datafile. Moreover, as the*second* argument***y*** at the procedure call**Aobj1** the separate symbol or their list can be. Besides that is supposed, before saving in a***m***–datafile***x*** all definitions of objects in the saved document should be evaluated in*separate* ***Input***–paragraphs on the basis of delayed assignments however existence of *headings* not required. Right there it should be noted that for ability of correct processing of the***m***–files created in the specified manner the predetermined ***$AobjNobj*** variable is used, that provides correct processing of the datafiles containing the procedures, in particular,**Aobj** and**Aobj1.** The next fragment represents source code of the**Aobj1** procedure along with the most typical examples of its usage.

In[2672]:= **Aobj1[x_ /; FileExistsQ[x] && StringTake[x, -2] == ".m", y_ /; SymbolQ[y] || ListQ[y] &&**

DeleteDuplicates[Map[SymbolQ[#] &, y]] == {True}] := Module[{a, c = "*)(*", d = $AobjNobj, k, t = {}, g = {}, h = Map[ToString, Flatten[{y}]], p, j = 1, v}, a = StringSplit[ReadFullFile[x], d][[2 ;;–1]]; a = Map[StringTake[#, {3,–3}] &, a]; For[j, j <= Length[h], j++, p = h[[j]];

For[k = 1, k <= Length[a], k++, If[SuffPref[a[[k]], Map[StringJoin[p, #] &, {"[", "=", ":"}], 1], AppendTo[t, StringReplace[a[[k]], c–> ""]]; AppendTo[g, p], Null]]]; v = {t, MinusList[h, g]};

If[v[[1]] != {}, ToExpression[v[[1]]];
Print["Software for " <> ToString[g] <> " is downloaded"], Null];
If[v[[2]] != {}, Print["Software for " <>
ToString[v[[2]]] <> " was not found"], Null]] In[2673]**:= Aobj1["Obj42.m", {Nobj90, Avz500, Agn67, Vsv47}]**

Software for {Nobj90, Avz500, Agn67, Vsv47} was not found In[2674]**:=**
Aobj1["Obj42.m", {Art1, Art2, Art3, Art4, Art, Agn, Avz, Rans,

IAN, Rae, Nobj }]
Software for {Art1,Art2,Art3,Art4,Art,Agn,Avz,Nobj} is downloaded Software for {Rans, IAN, Rae} was not found

There is a number of other rather interesting procedures for ensuring work with files of the**Mathematica*Input***–format whose names have extensions {***"nb"***,***"m"***, ***"txt"***}, etc. All such means are based on the basis of analysis of structure of the contents of files returned by access functions, in particular, **ReadFullFile.** Some of them gives a possibility to create the rather effective user libraries containing definitions of the**Mathematica–**objects. These and some other means have been implemented as a part of the special package supporting the releases***8 ÷ 10*** of the**Mathematica** system [48]. The part of these means will be considered in the present book slightly below. Certain remarks should be made concerning the**Save** function which saves the objects in a given file in the***Append***mode; at that, undefinite symbols in the datafile are not saved without output of any messages, i.e. the**Save** call returns*Null*, i.e. nothing. Meanwhile, at saving of a procedure or a function with a name***Avz*** in a datafile by means of the**Save** function in the datafile all active objects of the same name***Avz*** in the current session with different headings–*the identifiers of their originality*– are saved too. For elimination of this situation a generalization of the**Save** function concerning possibility of saving of objects with concrete headings is offered. So, the**Save1** procedure solves the given problem whose source code along with typical examples of its usage are represented by the following fragment.

In[2742] **:= A[x_] := x^2; A[x_, y_] := x+y; A[x_, y_, z_] := x+y+z; A[x__] := {x}; DefFunc3[A]**
Out[2742]= {"A[x_]:= x^2", "A[x_, y_]:= x+ y", "A[x_, y_, z_]:= x+ y+ z", "A[x__]:= {x}"}

In[2743]**:= Save1[x_ /; StringQ[x],**
y_ /; DeleteDuplicates[Map[StringQ, Flatten[{y}]]][[1]]] := Module[{Rs, t = Flatten[{y}], k = 1},

Rs[n_, m_] := Module[{b, c = ToString[Unique[b]], a = If[SymbolQ[m], Save[n, m], If[StringFreeQ[m, "["], $Failed,

StringTake[m, {1, Flatten[StringPosition[m, "["]][[1]]–1}]]]]}, If[a === Null, Return[], If[a === $Failed, Return[$Failed], If[SymbolQ[a], b = DefFunc3[a], Return[$Failed]]]]]; If[Length[b] == 1, Save[n, a], b = Select[b, SuffPref[#, m, 1] &]]; If[b != {}, b = c <> b[[1]], Return[$Failed]]; ToExpression[b]; a = c <> a; ToExpression["Save[" <> ToString1[n] <> "," <> ToString1[a] <> "]"]; BinaryWrite[n, StringReplace[ToString[StringJoin[Map[FromCharacterCode,

```
BinaryReadList[n]]]], c–> ""]]; Close[n]; ]; For[k, k <= Length[t], k++, Rs[x, t[[k]]]]]
In[2744] := Save1["rans_ian.m", {"A[x_, y_, z_]", "A[x__]"}]
In[2745]:= Clear[A]; DefFunc3[A]
Out[2745]= DefFunc3[A]
In[2746]:= << "rans_ian.m"
In[2747]:= B[x_] := x^2; DefFunc3[A]
Out[2747]= {"A[x_, y_, z_]:= x+ y+ z", "A[x__]:= {x}"}
In[2748]:= Agn = 67; Save1["Avz.m", {"A[x_, y_, z_]", "B", "A[x__]", "Agn"}]
In[2749]:= Clear[A, B, Agn]; Map[DefFunc3, {A, B, Agn}]
Out[2749]= {DefFunc3[A], DefFunc3[B], Agn}
In[2750]:= << "Avz.m"
Out[2750]= 67
In[2751]:= DefFunc3[A]
Out[2751]= {"A[x_, y_, z_]:= x+ y+ z", "A[x__]:= {x}"}
In[2752]:= {DefFunc3["B"], Agn}
Out[2752]= {{"B[x_]:= x^2"}, 67}
```

The procedure call **Save1[*x, y*]** saves in a datafile defined by the first factual argument***x,*** the definitions of the objects determined by the second factual argument***y*** *–the name of an active object in the current session or its heading in string format,or their combinations in the list form.* So, the**Save1** procedure can be used as the standard**Save** function, and solving a saving problem of the chosen objects activated in the current session in the datafile*differentially* on the basis of their headings. Thus, the successful procedure call returns*Null*, carrying out the demanded savings**;** otherwise**$Failed** or unevaluated call are returned. The previous fragment represents results of application of the **Save1** procedure for a selective saving in datafiles of the objects which have been activated in the**Mathematica** current session. In a number of cases the procedure**Save1** represents undoubted interest.
In a number of cases there is an urgent need of saving in a datafile of a state of the current session with possibility of its subsequent restoration by means of uploading of the datafile into current session different from the previous session. In this context, the**SaveCurrentSession** and**RestoreCS** procedures are rather useful for saving and restoration of a state of the current session respectively.So, the procedure call**SaveCurrentSession[]** saves a state of the **Mathematica** current session in the***m***–file***"SaveCS.m"*** with returning of the name of a target datafile. While the call**SaveCurrentSession[*x*]** saves a state of the**Mathematica** current session in a***m***–file***x*** with returning of the name of the target datafile***x;*** at that, if a datafile***x*** has not extension***"m"*** then this extension is added to the***x***–string. The procedure call**RectoreCS[]** restores the**Mathematica** current session that has been previously stored by means of the**SaveCurrentSession** procedure in datafile***"SaveCS.m"*** with returning the*Null***,** i.e. nothing. While the call**RectoreCS[*x*]** restores the**Mathematica** current session that has been previously stored by means of the procedure **SaveCurrentSession** in a***m***–datafile***x*** with returning the*Null***,** i.e. nothing. In absence of the above datafile the procedure call returns**$Failed.** The next fragment represents source codes of the above procedures along with typical examples of their usage.

```
In[2742]:= SaveCurrentSession[x___String] := Module[{a = Names["*"], b =
```

If[{x}== {}, "SaveCS.m", If[SuffPref[x, ".m", 2], x, x <> ".m"]]},

Save1[b, a]; b] In[2743]:= **SaveCurrentSession["Tallinn"]**
Out[2743]= "Tallinn.m"

In[2744]:= **RestoreCS[x___String] := Module[{a = If[{x}== {}, "SaveCS.m", If[FileExtension[x] == "m", x, $Failed]]}, If[a === $Failed, $Failed, On[General]; Quiet[Get[a]]; Off[General]]]**

In[2745] **:= RestoreCS["Tallinn.m"]**
In[2746]:= **RestoreCS["AvzAgnVsv.m"]**
Out[2746]= $Failed
So, the presented tools are rather useful in a case when is required to create copies of current sessions at certain moments of work with the system. The**DumpSave** function serves as other tool for saving of definitions of the objects in datafiles, creating datafiles of binary format that is optimized for input into the**Mathematica** system. Names of datafiles of this format have extension***"mx",*** and analogously to the previous format they can be loaded into the current session by the**Get** function. Unlike the**Save** function, the call of the**DumpSave** function returns the list of names and/or definitions of the objects saved in a***mx–***file. Meanwhile, it must be kept in mind a very essential circumstance that the datafiles created by means of the**DumpSave** function not only are most optimum for input into**Mathematica**, but also can't be loaded on a computing platform different from the platform on that they were created. Many interesting examples of application of the function **DumpSave** can be found in [30-33], some from them will be presented and a little below. Thus, it is necessary to work with datafiles of binary format only in the case when their usage in rather broad aspect isn't planned, i.e. in the sense this format has obviously internal character, without providing of the portability of the created means.

In a number of cases there is a necessity of loading into the current session of datafiles of types {***nb, m, mx, txt***} or datafiles of the***ASCII*** format without name extension which are located in one of directories of file system of the computer**;** moreover, having a full name of datafile we may not have certain information concerning its location in file system of the computer. In this context the**LoadFile** procedure solves this problem, whose source code and typical examples of its application the following fragment represents.

In[2575]:= **LoadFile[F_ /; StringQ[F]] := Module[{a, b, c}, If[! MemberQ[{"nb", "m", "mx", "txt", ""}, FileExtension[F]],**

Return["File <" <> F <> "> has an inadmissible type"], a = Flatten[{FindFile[F]}]; a = If[a === {$Failed}, SearchFile[F], a]; $Load$Files$ = a]; If[a == {}, Return["File <" <> F <> "> has not been found"], Quiet[Check[Get[$Load$Files$[[1]]], c = {$Failed}, {Syntax::sntxc, Syntax::sntxi}]]; If[c === {$Failed},

"File <" <> $Load$Files$[[1]] <> "> has inadmissible syntax", "File <" <> $Load$Files$[[1]] <> "> has been loaded; \n$Load$Files$ defines the list with full paths to the found files."], Return["File <" <> F <> "> has not been found"]]]
In[2576]:= **LoadFile["Obj42.m"]**
Out[2576]= "File<C:\aladjev\mathematica\Obj42.m> has been loaded**;**

$** Load$Files$ defines the list with full paths to the found files." In[2577]:= $Load$Files$**
Out[2577]= {"C:\users\aladjev\mathematica\Obj42.m"} In[2578]**:=**

LoadFile["AvzAgn.m"]
Out[2578]= "File<C:\Mathematica\AvzAgn.m> has been loaded;

$ Load$Files$ defines the list with full paths to the found files." In[2579]:= **$Load$Files$**
Out[2579]= {"C:\Mathematica\AvzAgn.m"}
In[2580]:= **LoadFile["Obj47.m"]**
Out[2580]= "File<Obj47.m> has not been found"

The procedure call **LoadFile[*w*]** uploads into the current session a data file given by its name***w*** and with an extension {***m,nb, mx, txt***} or at all without extension. Moreover, at finding of the list of datafiles with an identical name ***w*** uploading of the first of the list with return of the corresponding message is made, while thru the global**$Load$Files$** variable the procedure returns the list of all***w*** datafiles found in search process. The procedure processes the main erroneous and especial situations, including syntax of the found datafile, unacceptable for the**Get** function. In the case of lack of***w*** datafiles through the**$Load$Files$** variable the empty list, i.e. {} is returned. A rather simple and in certain cases the useful**MathematicaDF** procedure completes this section, its call**MathematicaDF[]** returns the list of***ListList*–** type, whose two–element members by the*first* elements contain type of the elements of**Mathematica** file system whereas by the*second* elements contain quantity of elements of this type. At that,***"NoExtension"*** defines datafiles without extension,***"Dir"*** defines directories while the others defines type of extension of a datafile. The following fragment represents source code of the **MathematicaDF** procedure along with a typical example of its application concerning the system**Mathematica*10.***

In[2982] **:= MathematicaDF[] := Module[{a = "Art26$Kr18$", b ={}, c = "", d}, Run["Dir " <> " /A/B/S " <> StrStr[$InstallationDirectory] <> " > " <> a]; While[! SameQ[c, EndOfFile], c = Read[a, String]; Quiet[If[DirectoryQ[c], AppendTo[b, "Dir"], If[FileExistsQ[c], d = FileExtension[c]; AppendTo[b, If[d === "", "NoExtension", d]], AppendTo[b, "NoFile"]]]]]; DeleteFile[Close[a]]; Map[{#[[1]], Length[#]}&, Gather[b, #1 === #2 &]]]** In[2983]:= **MathematicaDF[]**

Out[2983] = {{"CDCode", 1}, {"CreationID", 3}, {"PatchLevel", 1}, {"VersionID", 1}, {"Dir", 2076}, {"exe", 169}, {"m", 3477}, {"nb", 13355}, {"gen", 46}, {"NoExtension", 993}, {"cfs", 42}, {"jar", 168}, {"cmd", 2}, {"vbs", 2}, {"sh", 6}, {"pbs", 1}, {"json", 3}, == {"sln", 1}, {"cache", 3}, {"Cache", 1}, {"resources", 1}, {"vb", 1}, {"cl", 31}, {"py", 2}, {"so", 4}, {"jnilib", 2}, {"poly", 2}, {"node", 1}, {"cmap", 1}, {"cset", 1}, {"rws", 1}, {"kbd", 1}, {"ini", 1}, {"msg", 1}}

In[2984]:= **Plus[Sequences[Map[#[[2]] &, %]]]**
Out[2984]= 27 931

At last, the procedure call **OpSys[]** returns the type of operational platform. The procedure is useful in certain appendices above all of system character. The fragment represents source code of the procedure with an example.

In[5334]:= **OpSys[] := Module[{a = ToString[Unique["s"]], b}, Run["SystemInfo >" <> a]; b = StringTrim[StringTake[ReadList[a,**

String][[2]], {9,–1}]]; DeleteFile[a]; b] In[5335]:= **OpSys[]**

Out[5335]= **“**Microsoft Windows 7 Professional**”**

On that the representation of access means to the system files is completed, and means of operating with external datafiles will be presented in the next section. Meanwhile, the represented means of processing of system datafiles in a number of cases represent a quite certain interest, first of all, for various applications of the system character. So, certain means of the***AVZ_Package*** package use the above means [48].

7.2. Means of the*Mathematica*system for operating with external datafiles

According to such quite important indicator as means of access to datafiles the**Mathematica** system, in our opinion, possesses a number of advantages in comparison with the**Maple** system. First of all,**Mathematica** carries out automatic processing of hundreds of formats of data and their subformats on the basis of the unified usage of symbolical expressions. For each specific format the correspondence between internal and external representation of a format is determined, using the general mechanism of data elements of the **Mathematica** system. For today**Mathematica*10*** as a whole supports many various formats of datafiles for different purposes, their list can be received by means of the predetermined variables***$ImportFormats****(the imported files)* and***$ExportFormats****(the exported files)* in quantities***172*** and***144*** respectively. While the basic formats of datafiles are considered rather in details in [33].

By the function call **FileFormat[*x*]** an attempt to define an input format for a datafile given by a name***x*** in string format is made. In the case of existence for a datafile***x*** of name extension the**FileFormat** function is, almost, similar to the**FileExtension** function, returning the available extension, except for the case of packages*(**m**–datafiles)* when instead of extension the datafile type ***“Package”*** is returned. Meanwhile, in some cases the format identification is carried out incorrectly, in particular, the attempt to test a***doc***–file without an extension returns***“XLS”,*** ascribing it to the datafiles created by*Excel 95/ 97/2000/XP/2003* that is generally incorrect.

In[2557] **:= Map[FileFormat, {“AVZ_Package_1.nb”, “AVZ_Package_1.m”}]**
Out[2557]= {**”**NB**“, “**Package**”**}
In[2558]**:= FileFormat[“D:\AVZ_Package\Art1”]**
Out[2558]= **“**Text**”**
In[2559]**:= FileExtension[“D:\AVZ_Package\Art1”]**
Out[2559]= **””**
In[2560]**:= FileFormat[“Art1”]**

FileFormat **::**nffil**:** File not found during FileFormat[Art1]**.** >> Out[2560]= $Failed
In[2561]**:= FileFormat[“D:\AVZ_Package\AVZ_Package_1”]**

FileFormat **::**nffil**:** File not found during FileFormat[D**:**\AVZ_Package\AVZ_Package_1]**.** >> Out[2561]= **$**Failed
In[2562]**:= Map[FileFormat, {“C:/AVZ_P”, “C:/AVZ_P1”, “C:/Temp/Der”}]**
Out[2562]= {**”**NB**“, “**Package**“, “**XLS**”**}
In[2563]**:= FileFormat[“C:\Temp\Der.doc”]**
Out[2563]= **“**DOC**”**

In[2564]:= **FileFormat1[x_ /; StringQ[x]] := Module[**{**a**}**, If[FileExistsQ[x],** {**x, FileFormat[x]**}**, a = SearchFile[x]; If[a ==** {}, {}**, a = Map[**{**#, FileFormat[#]**}**&, a]; If[Length[a] == 1, a[[1]], a]]]]**

In[2565] **:= FileFormat1["AVZ_Package.m"]**
Out[2565]= {{"C**:**\Users\Mathematica\AVZ_Package.m"**,** "Package"},
{"C**:**\Temp\Mathematica\AVZ_Package.m"**,** "Package"},
{"C**:**\Mathematica\AVZ_Package.m"**,** "Package"},
{"D**:**\Temp\Mathematica\AVZ_Package.m"**,** "Package"}}
In[2566]**:= FileFormat1["Z123456789"]**
Out[2566]= {}
In[2567]**:= FileFormat1["C:\Users\Mathematica\AVZ_Package.m"]**
Out[2567]= {"C**:**\Users\Mathematica\AVZ_Package.m"**,** "Package"}
In[2610]**:= Map[FileFormat,** {**"C:/", "C:\"**}**]**
General**::**cdir**:** Cannot set current directory to$RL69B50**.** >> General**::**cdir**:** Cannot set current directory to$RRMBM4A**.**>>
===
General**::**stop**:** Further output of General**::**dirdep will be suppressed during this calculation**.** >>
Out[2610]= {"KML"**,** "KML"}

In[2611]**:= FileFormat2[x_ /; StringQ[x]] := Module[**{**a, b =** {}**, c, k = 1**}**, If[StringLength[x] == 3,**

If[MemberQ[{**":/", ":\"**}**, StringTake[x,–2]] && MemberQ[Adrive[], ToUpperCase[StringTake[x, 1]]], Return["Directory"], Null], If[DirectoryQ[x], Return["Directory"], a = SearchFile[x]];**

If[a == {}**, Return[**{}**], For[k, k <= Length[a], k++, c = a[[k]]; AppendTo[b,** {**c, FileFormat[c]**}**]]]]; If[Length[b] == 1, b[[1]], b]]**

In[2612] **:= Map[FileFormat2,** {**"C:/", "C:\", "C:/Temp", "C:\Temp"**}**]** Out[2612]= {"Directory"**,** "Directory"**,** "Directory"**,** "Directory"} In[2613]**:= FileFormat2["Obj47.m"]**
Out[2613]= {{"C**:**\Users\Aladjev\Mathematica\Obj47.m"**,** "Package"},

{"D**:**\AVZ_Package\Obj47.m"**,** "Package"}}

Moreover, by the function call**FileFormat[*x*]** an attempt to define the format of a datafile***x*** is made, that is located only in the subdirectories determined by***$Path*** variable otherwise returning***$Failed*** with print of the appropriate message as illustrates an example of the previous fragment. For elimination of similar situation the simple enough**FileFormat1** procedure is offered that expands the possibilities of the standard**FileFormat** function**,** and uses the **SearchFile** procedure which will be presented a little below. The procedure call**FileFormat1[*x*]** returns the simple or nested list,*first* element of a simple list defines the full path to a datafile***x*** while the*second* element**–** its format that is recognized by the**FileFormat** function**;** at that, the required datafile can be located in any directory of file system of the computer**;** absence of a datafile***x*** initiates the return of the empty list, i.e. {}. Moreover, at finding several datafiles with an identical name the nested list whose sublists have the specified format is returned. The previous fragment represents source code of the**FileFormat1** procedure along with typical examples of its usage. In some cases the**FileFormat1** procedure is more preferable than standard **FileFormat** function.

As it is noted above, the call **FileFormat[*F*]** tries to define an***Import***format for import of a datafile or*URL* corresponding to argument***F*;** meanwhile**,** on main directories of external memory*(disk,flash memory,etc.)* the call causes erroneous situation**;** at that, the function recognizes only the datafiles which are in the directories determined by***$Path*** variable**;** for elimination of the last situation the**FileFormat1** procedure above has been offered whereas the procedure can be quite simply extended for the purpose of elimination and the*first* situation. The**FileFormat2** procedure was programmed on the basis of the**FileFormat1** procedure, this procedure correctly processes the main directories, inaccessible or nonexistent devices of the external memory, and also datafiles from directories of file system of the computer. The previous fragment represents source code of the procedure with typical examples of its usage. Thus, earlier presented**FileFormat1** procedure provides check of format of the datafiles which are located in directories of file system of the computer irrespectively from their presence in the***$Path*** variable. While the **FileFormat2** procedure in addition correctly processes also main directories of external memory, bearing in mind the important circumstance that they are the key elements of file system of the computer. Indeed, examples of the previous fragment visually illustrate that the function call**FileFormat[*x*]** on main directory of a volume***x*** returns***"KML"*** format that is the*GIS* standard format which serves for storage of cartographical information instead of the ***Directory*** format. The call**FileFormat2[*x*]** eliminates this defect with return on similar objects***"Directory"*** while in other situations a call**FileFormat2[*x*]** is equivalent to a call**FileFormat1[*x*].**

At last, a version of the standard **FileFormat** function attempts to identify a datafile type without extension, being based on information of the creator of the datafile that is contained in the contents of the datafile. The**FileFormat3** procedure rather accurately identifies datafiles of the following often used types {***DOC, PDF, ODT, TXT, HTML***}. At that, concerning the***TXT*** type the verification of a datafile is made in the latter case, believing that the datafile of this type has to consist only of symbols with the following decimal codes**:**

0 ÷ 127 – ASCII*symbols*
1 ÷ 31 –*the control****ASCII****symbols*
32 ÷ 126 –*the printed****ASCII****symbols*
97 ÷ 122 –*letters of the Latin alphabet in the lower register*
129 ÷ 255 – Latin–1*symbols of****ISO***
192 ÷ 255 –*letters of the European languages*

The procedure call **FileFormat3[*x*]** returns the type of a datafile given by a name or a classifier***x*;** at that, if the datafile has an extension, it relies as the extension of the datafile. Whereas the call**FileFormat3[*x, y*]** with the second optional argument*–an arbitrary expression****y*** **–** in the case of datafile without extension returns its full name with the extension defined for it, at the same time renaming the datafile***x*,** taking into account the calculated format. The fragment below represents source code of the**FileFormat3** procedure along with the most typical examples of its usage.

In[2554]**:= FileFormat3[x_ /; FileExistsQ[x], t___] := Module[{b, c, a =**

```
FileExtension[x]}, If[a != "", ToUpperCase[a],
c = If[Quiet[StringTake[Read[x, String], {1, 5}]] === "%PDF–",

{ Close[x], "PDF"}[[–1]], Close[x]; b = ReadFullFile[x]; If[! StringFreeQ[b,
"MSWordDoc"], "DOC",
If[! StringFreeQ[b, ".opendocument.textPK"], "ODT", If[! StringFreeQ[b,
{"!DOCTYPE HTML ", "text/html"}], "HTML", If[MemberQ3[Range[0, 255],
DeleteDuplicates[Flatten[Map[

ToCharacterCode[#] &, DeleteDuplicates[Characters[b]]]]]], "TXT", Undefined]]]]];
If[{t}!= {}, Quiet[Close[x]]; RenameFile[x, x <> "." <> c], c]]]

In[2555] := Map[FileFormat3, {"C:\Temp.Burthday", "C:\Temp.cinema",
"C:/Temp/ransian", "C:/Temp/Book_Grodno", "C:/Temp/Math_Trials"}]
Out[2555]= {"DOC", "TXT", "HTML", "PDF", "DOC"}
In[2556]:= FileFormat3["C:/Temp/Math_Trials", 500]
Out[2556]= "C:\Temp\Math_Trials.DOC"
```

Using the algorithm implemented by the **FileFormat3** procedure it is rather simple to modify it for testing of other types of datafiles whose full names have no extension. That can be rather useful in the processing problems of the datafiles. In a certain relation the**Format3** procedure complements the standard**Format** function along with the procedures**Format1** and**Format2.**

The **Mathematica** provides effective enough system–independent access to all aspects of datafiles of any size. For ensuring operations of opening and closing of datafiles the following basic functions of access are used, namely**: OpenRead, OpenWrite, OpenAppend, Close.** Moreover, the name or full path to a datafile in string format acts as the only formal argument of the first three functions**;** at that, the function call**OpenWrite[]** without factual arguments is allowed, opening a new datafile located in the subdirectory intended for temporary files for writting. Whereas the**Close** function closes a datafile given by its name, full path or a***Stream*–**object. In attempt to close the closed or nonexistent file the system causes an erroneous situation. For elimination of such situation, undesirable in many cases, it is possible to use very simple**Closes** function providing the closing of any datafile including a closed or nonexistent datafile, without output of any erroneous messages with returning*Null***,** i.e. nothing, but, perhaps, the name or full path to the closed datafile, for example**:**

```
In[2351] := Close["D:/Math_myLib/test72.txt"]
General::openx: D:/Math_myLib/test72.txt is not open. >>
Out[2351]= Close["D:/Math_myLib/test72.txt"]

In[2352] := Closes[x_] := Quiet[Check[Close[x], Null]]
In[2353]:= Closes["D:/Math_myLib/test72.txt"]
```

An object of the following rather simple format is understood as the***Stream***- object of functions of access of**OpenRead, OpenWrite** and**OpenAppend:**
{*OutputStream*|*InputStream*}[<*Datafile*>, <*Logical IO channel*>]

By the function call **Streams[]** the list of***Stream***objects of datafiles opened in the current session including system files is returned. For obtaining the list of***Stream***-objects of datafiles**,**different from the system files it is possible to use the function call**StreamsU[].**

In[2642] **:= Streams[]**
Out[2642]= {OutputStream[”stdout“**,** 1]**,** OutputStream[”stderr“**,** 2]} In[2643]**:= S1 = OpenRead[“C:/Temp/Math_Trials.doc”]**
Out[2643]= InputStream[”C**:**/Temp/Math_Trials**.**doc“**,** 163]
In[2644]**:= S2 = OpenWrite[“C:\Temp/Book_Grodno.pdf”]** Out[2644]= OutputStream[”C**:**\Temp/Book_Grodno**.**pdf“**,** 164] In[2645]**:= Streams[]**
Out[2645]= {OutputStream[”stdout“**,** 1]**,** OutputStream[”stderr“**,** 2]**,**

InputStream[“C**:**/Temp/Math_Trials**.**doc“**,** 163]**,**
OutputStream[”C**:**\Temp/Book_Grodno**.**pdf“**,** 164]} In[2646]**:= OpenWrite[]**
Out[2646]= OutputStream[”C**:**\Users\Aladjev\AppData\Local\
Temp\m-e88a7f8c-339f-42e2-8da9-b6bd16e6cd50“**,** 165]

In[2647] **:= StreamsU[] := Select[Streams[], ! MemberQ[**{**“[stdout”, “[stderr”**}**, StringTake[ToString[#1],** {**13, 19**}**]] &]** In[2648]**:= StreamsU[]**
Out[2648]= {InputStream[”C**:**/Temp/Math_Trials**.**doc“**,** 163]**,**

OutputStream[“C**:**\Temp/Book_Grodno**.**pdf“**,** 164]**,**
OutputStream[”C**:**\Users\Aladjev\AppData\Local\ Temp\m-e88a7f8c-339f-42e2-8da9-b6bd16e6cd50“**,** 165]}

In[2649] **:= Close[“C:\Temp/Book_Grodno.pdf”]**
Out[2649]= “C**:**\Temp/Book_Grodno**.**pdf”
In[2650]**:= StreamsU[]**
Out[2650]= {InputStream[”C**:**/Temp/Math_Trials**.**doc“**,** 163]**,**

OutputStream[”C**:**\Users\Aladjev\AppData\Local\ Temp\m-e88a7f8c-339f-42e2-8da9-b6bd16e6cd50“**,** 165]} In[2658]**:= CloseAll[] := Map[Close, StreamsU[]]**
In[2659]**:= CloseAll[]**
Out[2659]= {”C**:**/Temp/Math_Trials**.**doc“**,** “C**:**\Users\Aladjev\

AppData \Local\Temp\m-e88a7f8c-339f-42e2-8da9-b6bd16e6cd50”} In[2660]**:= Streams[]**
Out[2660]= {OutputStream[”stdout“**,** 1]**,** OutputStream[”stderr“**,** 2]}

It must be kept in mind that after the termination of work with an opened datafile, it remains opened up to its obvious closing by the**Close** function. For closing of all channels and datafiles opened in the current session of the system, excepting system files, it is possible to apply quite simple**CloseAll** function, whose call**CloseAll[]** closes all mentioned open both channels and datafiles with return of the list of datafiles.

Similar to the **Maple** system the**Mathematica** system also has opportunity to open the same datafile on different streams and in various modes, using different coding of its name or path using alternative registers for letters or/ and replacement of separators of subdirectories“\” on”/“**,** and vice versa at opening of datafiles. The simple fragment below illustrates application of this approach for opening of the same datafile on two different channels on reading with the subsequent alternating reading of records from it.

In[2534]**:= F = “C:\Mathematica\AvzAgn”;** {**S, S1**}**=** {**OpenRead[F], OpenRead[If[UpperCaseQ[StringTake[F, 1]], ToLowerCase[F], ToUpperCase[F]]]**}

Out[2534] = InputStream[”C:\Mathematica\AvzAgn.m“, 118],
InputStream[”c:\mathematica\avzagn.m“, 119]}
In[2535]:= **t = {}; For[k = 1, k <= 3, k++, AppendTo[t, {Read[S], Read[S1]}]]**
Out[2535]= {”RANS1“, “RANS1“, “RANS2“, “RANS2“, “RANS3“, “RANS3”}

Meanwhile, it must be kept in mind that the special attention at opening of the same datafile on*different* channels is necessary and, above all, at various modes of access to the datafile in order to avoid of the possible especial and erroneous situations, including distortion of data in the datafile. Whereas in certain cases this approach at operating with large enough datafiles can give quite notable temporal effect along with simplification of certain algorithms of data processing which are in datafiles. The interesting enough examples of usage of the given approach can be found in our books [30-33].

Similar to the **Maple** system the**Mathematica** system has very useful means for work with the pointer defining the current position of scanning of a file. The following functions provide such work, namely**: StreamPosition, Skip, SetStreamPosition, Find.** The functions**StreamPosition, SetStreamPosition** allow to make monitoring of the current position of the pointer of an open datafile and to establish for it a new position respectively. Moreover, on the closed or nonexistent datafiles the calls of these functions cause erroneous situations. Reaction to the status of a datafile of the**Skip** function is similar, while the function call**Find** opens a stream on reading from a datafile. The sense of the presented functions is rather transparent and in more detail it is possible to familiarize with them, for instance, in [30,33]. In connection with the told arises the question of definition of the status of a datafile**–** opened, closed or doesn’t exist. In this regard the**FileOpenQ** procedure can be quite useful, whose source code with examples of application represents the next fragment together with an example of usage of the standard**Skip** function.

In[2542] **:= FileOpenQ[F_ /; StringQ[F]] := Module[{A, a = FileType[F], b, d, x = inputstream, y = outputstream, c = Map[ToString1, StreamsU[]], f = ToLowerCase[StringReplace[F, “\”–> “/”]]}, A[x_] := Module[{a1 = ToString1[x],**

b1 = StringLength[ToString[Head[x]]]}, ToExpression[“{” <> StrStr[Head[x]] <> “,” <>

StringTake[a1, {b1 + 2,–2}] <> “}“]]; If[MemberQ[{Directory, None}, a], Return[$Failed], Clear[inputstream, outputstream]; d = ToExpression[ToLowerCase[StringReplace[ToString1[Map[A, StreamsU[]]], “\\”–> “/”]]]; a = Select[d, #[[2]] === f &]; If[a == {}, {inputstream, outputstream}= {x, y}; False, a = {ReplaceAll[a, {inputstream–> “read”, outputstream–> “write”}], {inputstream, outputstream}={x, y}}[[1]]]]; If[Length[a] == 1, a[[1]], a]]

In[2543] **:= OpenRead[“C:\Temp\cinema.txt”]; Write[“rans.ian”]; Write[“C:\Temp/Summ.doc”]; Write[“C:\Temp/Grin.pdf”]**
In[2544]**:= Map[FileOpenQ, {“rans.ian”, “C:\Temp\Grin.pdf”, “C:\Temp/Summ.doc”, “C:\Temp/cinema.txt”}]**
Out[2544]= {{”write“, “rans.ian“, 85}, {”write“, “c:/temp/grin.pdf“, 87}, {”write“, “c:/temp/summ.doc“, 86}, {”read“, “c:/temp/cinema.txt“, 84}}
In[2545]**:= Map[FileOpenQ, {“C:\Temp\Books.doc”, “C:\Books.doc”}]**

Out[2545]= {{}, $Failed}

The procedure call **FileOpenQ[*F*]** returns the nested list {{***R, F, Channel***},**…**} if a datafile***F*** is open on reading/writing*(**R**=* {***"read"**|**"write"***}*)*,***F*** defines actually the datafile***F*** in the stylized format*(LowerCase+all"\"are replaced on"/")* while***Channel*** defines the logical channel on which the datafile***F*** in the mode specified by the first element of the list***R*** was open**;** if datafile***F*** is closed, the empty list is returned, i.e. {}**,** if datafile***F*** is absent, then***$Failed*** is returned. At that, the nested list is used with the purpose, that the datafile***F*** can be opened according to syntactically various file specifiers, for example, **"**Agn47**"** and**"**AGN47**"**, allowing to carry out its processing in the different modes simultaneously.
The**FileOpenQ1** procedure is an useful enough extension of the**FileOpenQ** procedure considered above. The procedure call**FileOpenQ1[*F*]** returns the nested list of the format {{***R, x, y,…,z***}, {{***R, x1, y1,…,z1***}} if a datafile***F*** is open for reading or writing*(**R** =* {***"in"**|**"out"***}*)*, and***F*** defines the datafile in any format*(Register+**"/"*** and/or**"**\"**)**; if the datafile***F*** is closed or is absent, the empty list is returned, i.e. {}. Moreover, sublists {***x,y, …, z***} and {***x1, y1, …, z1***} define datafiles or full paths to them that are open for reading and writing respectively. Moreover, if in the current session all user datafiles are closed, except system files, the call**FileOPenQ1[*x*]** on an arbitrary string***x*** returns **$Failed.** The datafiles and paths to them are returned in formats which are defined in the list returned by the function call**Streams[],**irrespective of the format of the datafile***F.*** The following fragment presents source code of the **FileOpenQ1** procedure along with typical examples of its usage.

In[2615] **:= FileOpenQ1[F_ /; StringQ[F]] := Module[{a = StreamFiles[], b, c, d, k = 1, j}, If[a === "AllFilesClosed", Return[False], c = StringReplace[ToLowerCase[F], "/"–> "\"]; b = Mapp[StringReplace, Map[ToLowerCase, a], "/"–> "\"]]; For[k, k <= 2, k++, For[j = 2, j <= Length[b[[k]]], j++, If[Not[SuffPref[b[[k]][[j]], c, 2] || SuffPref[b[[k]][[j]], "\" <> c, 2]], a[[k]][[j]] = Null; Continue[], Continue[]]]]; b = Mapp[Select, a, ! # === Null &]; b = Select[b, Length[#] > 1 &]; If[Length[b] == 1, b[[1]], b]]**

In[2616] **:= OpenWrite["Kherson.doc"]; OpenWrite["C:/Temp/Books.doc"]; OpenWrite["RANS"]; OpenRead["C:/Temp\Cinema.txt"]; Read["C:/Temp\Cinema.txt", Byte];**

In[2617] **:= CloseAll[]; FileOpenQ1["AvzAgnArtKrSv"]**
Out[2617]= False
In[2618]**:= Map[FileOpenQ1, {"Kherson.doc", "C:/Temp\Books.doc",**

"RANS", "C:/Temp\Cinema.txt", "Agn"}] Out[2618]= {{"out**"**, **"**Kherson.doc"}, {"out**"**, **"**C:/Temp\Books.doc"}, {"out**"**, **"**RANS"}, {"in**"**, **"**C:/Temp\Cinema.txt"}, {}}
In[2619]**:= Map[FileOpenQ1, {"Kherson.doc", "C:/Temp\Books.doc", "RANS", "C:/Temp\Cinema.txt", "Agn"}]** Out[2619]= {{"write**"**, **"**kherson.doc**"**, 1458}, {"write**"**, **"**c:/temp/books.doc**"**, 1459}, {"write**"**, **"**rans**"**, 1460}, {"read**"**, **"**c:/temp/cinema.txt**"**, 1461}, $Failed}

So, functions of access **Skip, Find, StreamPosition** and**SetStreamPosition** provide quite effective tools for rather thin manipulation with datafiles and in combination with a

number of other functions of access they provide the user with the standard set of functions for processing of datafiles, and give opportunity on their base to create own tools allowing how to solve specific problems of work with datafiles, and in a certain degree to extend standard opportunities of the system. A number of similar means is presented and in the present book, and in our***AVZ_Package*** package [48]. In addition to the represented standard operations of datafiles processing, a number of other means of the package rather significantly facilitates effective programming of higher level at the solution of many problems of datafiles processing and management of the system. Naturally, the consideration rather in details of earlier presented tools of access to*datafiles* and the subsequent tools doesn**'**t enter purposes of the present book therefore we will present relatively them only short excursus in the form of a brief information with some comments on the represented means.
Among standard means of the datafiles processing, the following functions can be noted, namely**: FileNames–** depending on the coding format returns the list of full paths to the datafiles and/or directories contained in the given directory onto arbitrary nesting depth in file system of the computer. While the functions**CopyFile, RenameFile, DeleteFile** serve for*copying***,***renaming* and*removal* of the given datafiles accordingly. Except listed means for work with datafiles the**Mathematica** has a number of rather useful functions that here aren**'**t considered**but** with which the interested reader can familiarize in reference base of the system or in the corresponding literature [55,60,64]. Along with the above functions**OpenRead, OpenWrite, Read, Write, Skip** and**Streams** of the lowest level of access to datafiles**,** the functions**Get, Put, Export,Import, ReadList, BinaryReadList, BinaryWrite, BinaryRead** are not less important for support of access to datafiles which support operations of reading and writing of data of the required format. With these means along with a whole series of rather interesting examples and features of their use, at last with certain critical remarks to their address the reader can familiarize in [30-33]. Meantime, these tools of access together with already considered means and means remaining without our attention form a rather developed system of effective processing of datafiles of various formats.

On the other hand, along with actually processing of the internal contents of datafiles, the**Mathematica** has a number of means for search of files, their testing, work with their names, etc. We will list only some of them, namely**: FindFile, FileExistsQ, FileNameDepth, FileNameSplit, ExpandFileName, FileNameJoin, FileBaseName, FileNameTake.** With the given means along with a whole series of rather interesting examples and features of their use**,** at last with certain critical remarks to their address the reader can familiarize in [30-33]. In particular, as it was noted earlier, the**FileExistsQ** function like some other functions of access in the course of search is limited only to the directories defined in the predetermined***$Path*** variable. For the purpose of elimination of this shortcoming simple enough**FileExistsQ1** procedure has been offered. The next fragment represents source code of the**FileExistsQ1** procedure along with typical examples of its usage.

In[2534]**:= FileExistsQ1[x__ /; StringQ[{x}[[1]]]] := Module[{b = {x}, a = SearchFile[{x}[[1]]]}, If[a == {}, False, If[Length[b] == 2 && ! HowAct[b[[2]]], ToExpression[ToString[b[[2]]] <>" =" <>ToString1[a]], Null]; True]]** In[2535]**:= {FileExistsQ1["Mathematica.doc", t], t}**

Out[2535]= {True**,** {**”**C:\Mathematica\Mathematica.doc**“,**

“ E**:**\Mathematica\Mathematica.doc**”**}}
In[2536]**:= FileExistsQ[“Books.doc”]**
Out[2536]= False
In[2537]**:= FileExistsQ1[“Books.doc”]**
Out[2537]= True
In[2538]**:= FileExistsQ1[“Book_avz.doc”]**
Out[2538]= False

The procedure call **FileExistsQ1[*x*]** with one actual argument returns*True* if ***x*** determines a datafile, really existing in the file system of the computer and *False* otherwise**;** whereas the call**FileExistsQ1[*x*, *y*]** in addition through the actual argument***y*** *–an undefinite variable***–** returns the list of full paths to the found datafile***x*** if the main result of the call is*True***.**

The previous procedure enough essentially uses the **SearchFile** procedure providing search of the given datafile in file system of the computer. At that, the procedure call**SearchFile[*f*]** returns the list of paths to a datafile***f*** found within file system of the computer**;** in the case of absence of the required file ***f*** the procedure call**SearchFile[*f*]** returns the empty list, i.e. {}. We will note, the procedure**SearchFile** essentially uses the standard**Run** function of the **Mathematica** system that is used by a number of tools of our***AVZ_Package*** package [48]. The fragment below represents source code of the**SearchFile** procedure along with typical examples of its usage.

In[2532]**:= SearchFile[F_ /; StringQ[F]] := Module[**{**a, b, f, dir, h = StringReplace[ToUpperCase[F], “/”–> “\”]**}**,** {**a, b, f**}**=** {**Map[ToUpperCase[#] <> “:\” &, Adrive[]],** {}**,**

ToString[Unique[“d”]] <> “.txt” }**; dir[y_ /; StringQ[y]] := Module[**{**a, b, c, v**}**, Run[“Dir ” <> “/A/B/S ” <> y <> ” > ” <> f]; c =** {}**; Label[b];**

a = StringReplace[ToUpperCase[ToString[v = Read[f, String]]], “/”–> “\”]; If[a == “ENDOFFILE”, Close[f]; DeleteFile[f];

Return[c], If[SuffPref[a, h, 2], If[FileExistsQ[v], AppendTo[c, v]]; Goto[b], Goto[b]]]]]; For[k = 1, k <= Length[a], k++, AppendTo[b, dir[a[[k]]]]]; Flatten[b]]

In[2533]**:= SearchFile[“AVZ_Package.nb”]**
Out[2533]= {**”**C**:**\Users\Aladjev\Mathematica\AVZ_Package.nb**“,**

“ E**:**\AVZ_Package\AVZ_Package.nb**”**}
In[2534]**:= SearchFile[“init.m”]**
Out[2534]= {**”**C**:**\Program Files\Wolfram Research\Mathematica\10.0

\ AddOns\Applications\AuthorTools\Kernel\init.m**“,**
== **“**C**:**\Users\All Users\Mathematica\Kernel\init.m**”**}

In[2535] **:= Length[%]**
Out[2535]= 100
In[2536]**:= SearchFile[“Mathematica.doc”]**

```
Out[2536]= {”C:\Mathematica\Mathematica.doc“,
“ E:\Mathematica\Mathematica.doc”}
In[2537]:= SearchFile[“AVZ_AGN_VSV_ART_KR.590”]
Out[2537]= {}
In[2538]:= SearchFile[“Cinema.txt”]
Out[2538]= {”C:\Temp\Cinema.txt”}
In[2587]:= SearchFile1[x_ /; StringQ[x]] := Module[{a, b, c, d, f = {}, k = 1},
If[PathToFileQ[x], If[FileExistsQ[x], x, {}], a = $Path; f
=Select[Map[If[FileExistsQ[# <> “\” <> ToUpperCase[x]], #,
“Null”] &, a], # != “Null” &]; If[f != {}, f, d = Map[# <> “:\” &, Adrive[]]; For[k, k
<= Length[d], k++, a = Quiet[FileNames[“*”, d[[k]], Infinity]]; f = Join[f,
Select[Map[If[FileExistsQ[#] && SuffPref[ToUpperCase[#], “\” <>ToUpperCase[x],
2], #, “Null”]&, a], # != “Null” &]]]; If[f == {}, {}, f]]]]
In[2588] := SearchFile1[“BirthDay.doc”]
Out[2588]= {”C:\Temp\Birthday.doc“,
“E:\ARCHIVE\MISCELLANY\ Birthday.doc“, “E:\Temp\Birthday.doc”}
In[2589]:= SearchFile1[“Cinema.txt”]
Out[2589]= {”C:\Program Files\Wolfram Research\Mathematica\10.0
\SystemFiles\Links”}
In[2590]:= SearchFile1[“C:\Mathematica\Tuples.doc”]
Out[2590]= “C:\Mathematica\Tuples.doc”
```

The **SearchFile1** procedure which is a functional analog of the**SearchFile** procedure completes the previous fragment. The call**SearchFile[*F*]** returns the list of full paths to a datafile***F*** found within file system of the computer**;** in the case of absence of the required file***F*** the procedure call**SearchFile[*F*]** returns the empty list, i.e. {}. Unlike the previous procedure the**SearchFile1** procedure seeks out a datafile in***3*** stages**:*(1)*** if the required datafile is given by the full path only*existence* of the concrete datafile is checked, at detection the full path to it is returned,***(2)*** search is done in the list of the directories determined by the predetermined***$Path*** variable,***(3)*** search is done within all file system of the computer. The procedure**SearchFile1** essentially uses the procedure**Adrive** that is used by a number of our means of access [48]. It should be noted that speed of both procedures generally very essentially depends on the sizes of file system of the computer, first of all, if a required datafile isn’t defined by the full path and isn’t in the directories determined by the***$Path*** variable. Moreover, in this case the search is done even in the *Windows***“C:\$Recycle.Bin”** directory.

Along with means of processing of external datafiles the system has also the set of useful enough means for manipulation with directories of both the **Mathematica,** and file system of the personal computer in general. We will list only some of these important functions, namely**:**
DirectoryQ[*D*]–*the call returns True if a string***D***defines an existing directory, and False otherwise***;***unfortunately***,***the standard procedure at coding***”/”***at the end of the string***D***returns False irrespective of existence of the tested directory***;***a quite simple***DirQ***procedure eliminates the defect of this standard means.*

In[2602]:= **DirQ[d_ /; StringQ[d]] := DirectoryQ[StringReplace[d, "/"–> "\"]]**

In[2603] := **Map1[{DirectoryQ, DirQ}, {"C:/Mathematica\"}]** Out[2603]= {True, True}
In[2604]:= **Map1[{DirectoryQ, DirQ}, {"C:/Mathematica/"}]**
Out[2604]= {False, True}
In[2605]:= **Map1[{DirectoryQ, DirQ}, {"C:/Mathematica"}]**
Out[2605]= {True, True}

DirectoryName[*W***]–***the call returns a path to a directory containing datafile***W;** *moreover,ifWis a real subdirectory,the chain of subdirectories to it is returned;at that,taking into account the file concept that identifies datafiles and subdirectories, and the circumstance that the call***DirectoryName[***W***]***doesn't consider the actual existence of***W,***similar approach in a certain measure could be considered justified, but on condition of taking into account of reality of a tested path***W***such approach causes certain questions. Therefore from this standpoint a quite simple***DirName** *procedure which returns***"None"***if***W***is a subdirectory,the path to a subdirectory containing datafile***W,***and***$Failed***otherwise is offered. Moreover,search is done within all file system of the computer,but not within only system of subdirectories determined by the predetermined***$Path***variable.*

In[2605]:= **DirName[F_/; StringQ[F]] := If[DirQ[F], "None",
If[! FileExistsQ1[F], $Failed, Quiet[Check[FileNameJoin[FileNameSplit[F][[1;–2]]], "None"]]]]**

In[2606] := **Map[DirectoryName, {"C:/Temp/Cinema.txt", "C:/Temp"}]** Out[2606]= {"D:\MathMyLib\", "D:\"}
In[2607]:= **Map[DirName, {"C:/Temp/Cinema.txt", "C:/Temp", "G:\"}]** Out[2607]= {"Temp", "None", $Failed}

CreateDirectory[*d***]–***the call creates the given directory***d***with return of the path to it;meanwhile this tool doesn't work in the case of designation of the nonexistent device of external memory (disk,flash card,etc.) therefore we created a rather simple* **CDir***procedure which resolves this problem:the procedure call***CDir[***d***]***creates the given directory***d***with return of the full path to it;in the absence or inactivity of the device of external memory the directory is created on a device from the list of active devices of external memory that has the maximal volume of available memory with returning of the full path to it:*

In[2612]:= **CDir[d_ /; StringQ[d]] := Module[{a},
Quiet[If[StringTake[d, {2, 2}] == ":", If[MemberQ[a, StringTake[d, 1]],**

CreateDirectory[d], a = Adrive[]; CreateDirectory[Sort[a, FreeSpaceVol[#1] >=FreeSpaceVol[#2] &][[1]]<> StringTake[d, {2,–1}]]], CreateDirectory[d]]]]

In[2613] := **CreateDirectory["G:\Temp\GSV/ArtKr"]**
CreateDirectory::nffil: File not found during CreateDirectory… >>
Out[2613]= $Failed
In[2614]:= **CDir["G:\Temp\GSV/ArtKr"]**
Out[2614]= "G:\Temp\GSV\ArtKr"
In[2615]:= **CDir["A:/Temp\AVZ\Tallinn\IAN\Grodno/Kherson"]**

Out[2615]= "C:\Temp\AVZ\Tallinn\IAN\Grodno\Kherson"

CopyDirectory* [*d1, d2*]–the function call completely copies a**d1**directory into a **d2**directory,however in the presence of the accepting directory**d2**the function call* **CopyDirectory[*d1,d2*]***causes an erroneous situation with return of**$Failed**that in a number of cases is undesirable.For the purpose of elimination of such situation a rather simple*CopyDir*function can be offered,which in general is similar to the standard*CopyDirectory*function,but with the difference that in the presence of the accepting directory**d2**the**d1**directory is copied as a subdirectory of the**d2**with returning of the full path to it,for example:*

In[2625] **:= CopyDirectory["C:/Mathematica", "C:/Temp"]**
CopyDirectory::filex: Cannot overwrite existing file C:/Temp. >>
Out[2626]= $Failed

In[2626]**:= CopyDir[d_ /; StringQ[d], p_ /; StringQ[p]] := CopyDirectory[d, If[DirQ[p], p <> "\" <> FileNameSplit[d][[–1]], p]]** In[2627]**:= CopyDir["C:/Mathematica", "C:/Temp"]**
Out[2627]= "C:\Temp\Mathematica"

DeleteDirectory[*W*]–*the call deletes from file system of the computer the given directory**W**with return Null,i.e. nothing,regardless of attributes of the directory (Archive,Read–only,Hidden,System).Meanwhile,such approach,in our opinion, isn't quite justified,relying only on the circumstance that the user is precisely sure that he deletes.While in the general case there has to be an insurance from removal, for example,of the datafiles and directories having such attributes as Read-only (**R**), Hidden (**H**) and System (**S**).To this end,for example,it is possible before removal of an element of file system to previously check up its attributes what useful enough* **Attrib***procedure considered in the following section provides.*

The reader can familiarize with other useful enough means of processing of datafiles and directories in reference base on the**Mathematica** system and, in particular, in such editions, as [28-33,51-53,60,62,64,71].

7.3. Means of the*Mathematica*system for processing of attributes of directories and datafiles

The **Mathematica** system has no means for work with attributes of datafiles and directories what, in our opinion, is a rather essential shortcoming, first of all, at creation on its basis of various data processing systems. By the way, similar means are absent also in the**Maple** system therefore we created for it a set of procedures {***Atr,F_atr, F_atr1, F_atr2***} [47] which have solved the given problem. The means represented below solve the similar problem for the**Mathematica** system too. The following fragment represents the**Attrib** procedure providing processing of attributes of datafiles and directories.

In[2670] **:= Attrib[F_ /; StringQ[F], x_ /; ListQ[x] && DeleteDuplicates[Map3[MemberQ, {"–A", "–H", "–S", "–R", "+A", "+H", "+S", "+R"}, x]] == {True}|| x == {}||x == "Attr"] := Module[{a, b = "attrib ", c, d = " > ", h = "attrib.exe", p, f, g, t, v},**

```
a = ToString[v = Unique["ArtKr"]]; If[Set[t, LoadExtProg["attrib.exe"]] === $Failed, Return[$Failed], Null]; If[StringLength[F] == 3 && DirQ[F] &&

StringTake[F, {2, 2}] == ":", Return["Drive " <> F], If[StringLength[F] == 3 && DirQ[F], f = StandPath[F],
If[FileExistsQ1[StrDelEnds[F, "\", 2], v], g = v;
f = StandPath[g[[1]]]; Clear[v],

Return["<" <> F <> "> is not a directory or a datafile"]]]]; If[x === "Attr", Run[b <> f <> d <> a],
If[x === {}, Run[b <> "–A–H–S–R " <> f <> d <> a],

Run[b <> StringReplace[StringJoin[x], {"+"–> " +", "–"–> "–"}] <> " " <> f <> d <> a]]];

If[FileByteCount[a] == 0, Return[DeleteFile[a]],
d = Read[a, String]; DeleteFile[Close[a]]]; h = StringSplit[StringTrim[StringTake[d,

{ 1, StringLength[d]–StringLength[f]}]]]; Quiet[DeleteFile[t]]; h = Flatten[h /.
{"HR"–> {"H", "R"}, "SH"–> {"S", "H"}, "SHR"–> {"S", "H", "R"}, "SRH"–>
{"S", "R", "H"}, "HSR"–> {"H", "S", "R"}, "HRS"–> {"H", "R", "S"}, "RSH"–>
{"R", "S", "H"}, "RHS"–> {"R", "H", "S"}}];

If[h === {"File", "not", "found", "–"}||
MemberQ[h, "C:\Documents"], "Drive " <> f, {h, g[[1]]}]]

In[2671] := Attrib["C:\Temp\Cinema.txt", {"+A", "+S", "+R"}] In[2672]:=
Attrib["Cinema.txt", {"+A", "+S", "+R"}]
In[2673]:= Attrib["C:\Temp\Cinema.txt", "Attr"]
Out[2673]= {{"A", "S", "R"}, "C:\Temp\Cinema.txt"}
In[2674]:= Attrib["Cinema.txt", "Attr"]
Out[2674]= {{"A", "S", "R"}, "C:\Program Files\Wolfram Research\

Mathematica \10.0\Cinema.txt"} In[2675]:= Attrib["C:\Temp\Cinema.txt", {}]
In[2676]:= Attrib["C:\Temp\Cinema.txt", "Attr"]
Out[2676]= {{}, "C:\Temp\Cinema.txt"}
In[2677]:= Attrib["C:\", "Attr"]
Out[2677]= "Drive C:\"
In[2678]:= Attrib["G:\", "Attr"]
Out[2678]= "<G:\> is not a directory or a datafile"
In[2679]:= Attrib["RANS.IAN", "Attr"]
Out[2679]= {{"A"}, "C:\Users\Aladjev\Documents\rans.ian"}

In[2680] := Attrib["RANS.IAN", {"+A", "+S", "+H", "+R"}]
In[2681]:= Attrib["RANS.IAN", "Attr"]
Out[2681]= {{"A", "S", "H", "R"},

" C:\Users\Aladjev\Documents\rans.ian"} In[2682]:= Attrib["RANS.IAN", {"–S", "–R", "–H"}]
In[2683]:= Attrib["RANS.IAN", "Attr"]
Out[2683]= {{"A"}, "C:\Users\Aladjev\Documents\rans.ian"} In[2684]:=
Attrib["c:/temp\", "Attr"]
```

Out[2684]= {{}, {"C:\Temp"}}
In[2685]:= **Attrib["c:/temp\", {"+A"}]**
In[2686]:= **Attrib["c:/temp\", "Attr"]**
Out[2686]= {{"A"}, "C:\Temp"}

The successful procedure call **Attrib[*f*,"*Attr*"]** returns the list of attributes of a given datafile or directory***f*** in the context***Archive("A"), Read–only("R"), Hidden("H")*** and***System("S").*** At that, also other attributes inherent to the system datafiles and directories are possible**;** thus, in particular, on the main directories of devices of external memory***"Drive f",*** while on a nonexistent directory or datafile the message***"f isn't a directory or datafile"*** is returned. At that, the call is returned in the form of the list of the format {***x, y, …, z, F***} where the last element determines a full path to a datafile or directory***f;*** the datafiles and subdirectories of the same name can be in various directories, however processing of attributes is made only concerning the first datafile/ directory from the list of the objects of the same name. If the full path to a datafile/directory***f*** is defined as the first argument of the**Attrib** procedure, specifically only this object is processed. The elements of the returned list that precede its last element determine attributes of a processed directory or datafile. The procedure call**Attrib[*f,* {}]** returns*Null***,** i.e. nothing, canceling all attributes for a processed datafile/directory***f*** whereas the procedure call **Attrib[*f,* {*"x", "y",…, "z"*}]** where***x, y, z***∈{**"–A", "–H", "–S", "–R", "+A", "+H", "+S", "+R"**}**,** also returns*Null***,** i.e. nothing, setting/cancelling the attributes of the processed datafile/directory***f*** determined by the second argument. At impossibility to execute processing of*attributes* the procedure call**Attrib[*f*,*x*]** returns the corresponding messages. The**Attrib** procedure allows to carry out processing of attributes of both the file, and the directory located in any place of file system of the computer. This procedure is represented to us as a rather useful means for operating with file system of the computer. In turn, the following**Attrib1** procedure in many respects is similar to the **Attrib** procedure both in the functional, and in the descriptive relation, but the**Attrib1** procedure has certain differences. The successful procedure call **Attrib[*f*,"*Attr*"]** returns the list of attributes in string format of a directory or datafile***f*** in the context***Archive*** (**"A"**)**,*Read-only*** (**"R"**)**,*Hidden*** (**"H"**)**,*System*** (**"S"**)**.** The procedure call**Attrib1[*f*,**{}**]** returns*Null***,** i.e. nothing, canceling all attributes for the processed datafile/directory***f*** whereas the procedure call **Attrib1[*f*,**{***"x", "y",…, "z"***}**]** where***x, y, z***∈{**"–A", "–H", "–S", "–R", "+A", "+H", "+S", "+R"**}**,** also returns*Null***,** i.e. nothing, setting/cancelling the attributes of the processed datafile/directory***f*** determined by the second argument, while call**Attrib1[*f, x, y*]** with the***3rd*** optional argument***y*** *–an expression–* in addition deletes the program file***"attrib.exe"*** from the directory determined by the call**Directory[].** The following fragment represents source code of the **Attrib1** procedure along with the most typical examples of its usage.

In[2670] **:= Attrib1[F_ /; StringQ[F], x_ /; ListQ[x] && DeleteDuplicates[Map3[MemberQ,{"–A", "–H", "–S", "–R", "+A", "+H", "+S", "+R"}, x]] == {True}||x == {}|| x == "Attr", y___] := Module[{a = "$ArtKr$", b = "attrib ", c, d = " > ", h = "attrib.exe",**

p, f, g = Unique["agn"] }, If[LoadExtProg["attrib.exe"] === $Failed, Return[$Failed], Null]; If[StringLength[F] == 3 && DirQ[F] && StringTake[F, {2, 2}] == ":",

Return[“Drive ” <> F], If[StringLength[F] == 3 && DirQ[F], f = StandPath[F], If[FileExistsQ1[StrDelEnds[StringReplace[F, “/” –> “\”], “\”, 2], g];

f = StandPath[g[[1]]]; Clear[g],

Return[“<” <> F <> “> is not a directory or a datafile”]]]]; If[x === “Attr”, Run[b <> f <> d <> a],
If[x === {}, Run[b <> “–A–H–S–R ” <> f <> d <> a],

Run[b <> StringReplace[StringJoin[x], {“+”–> ” +”, “–”–> “–”}] <> ” ” <> f <> d <> a]]];

If[FileByteCount[a] == 0, Return[DeleteFile[a]],
d = Read[a, String]; DeleteFile[Close[a]]]; h = StringSplit[StringTrim[StringTake[d, {1, StringLength[d]–StringLength[f]}]]]; Quiet[DeleteFile[f]];

If[{y}!= {}, DeleteFile[Directory[] <> “\” <> “attrib.exe”], Null]; h = Flatten[h /. {“HR”–> {“H”, “R”}, “SH”–> {“S”, “H”}, “SHR”–> {“S”, “H”, “R”}, “SRH”–> {“S”, “R”, “H”}, “HSR”–> {“H”, “S”, “R”}, “HRS”–> {“H”, “R”, “S”}, “RSH”–> {“R”, “S”, “H”}, “RHS”–> {“R”, “H”, “S”}}];

If[h === {“File”, “not”, “found”, “–”}|| MemberQ[h, “C:\Documents”], “Drive ” <> f, h]]

In[2671] **:= Mapp[Attrib1, {“C:/tmp/a b c”, “C:/tmp/I a n.doc”}, {“+A”, “+R”}]**
Out[2671]= {Null, Null}
In[2672]**:= Mapp[Attrib1, {“C:/tmp/a b c”, “C:\tmp\I a n.doc”}, “Attr”]** Out[2672]= {{”A“, “R”}, {”A“, “R”}}
In[2673]**:= Attrib1[“G:\Temp\Cinema.txt”, “Attr”]**
Out[2673]= {”A“, “S“, “R”}
In[2674]**:= Attrib1[“G:\Temp\Cinema.txt”, {}]**
In[2675]**:= Attrib1[“G:\Temp\Cinema.txt”, “Attr”]**
Out[2675]= {}

Both procedures essentially use our procedures **LoadExtProg, StrDelEnds, StandPath, FileExistsQ1** and**DirQ** along with usage of the standard**Run** function and the***Attrib*** function of the***MS DOS*** operating system. At that, the possibility of removal of the***“attrib.exe”*** program file from the directory which is defiined by the call**Directory[]** after a call of the**Attrib1** procedure leaves file**Mathematica** system unchanged. So, in implementation of both procedures the system**Run** function was enough essentially used, that has the following coding format, namely**:**

Run[***s1,…, sn*****]–***in the basic operational system (for example,* ***MS DOS****) executes a command formed from expressions****sj(j=1..n****) which are parted by blank symbols with return of code of success of the command completion in the form of an integer.* *As a rule,the****Run****function doesn’t demand of an interactive input,but on certain operational platforms it generates text messages.To some extent the****Run****function is similar to the functions* {***system, ssystem***}*of the****Maple****system.In* [33]*rather interesting examples of application of the****Run****for performance in the environment of the****Mathematica****with the****MS DOS****commands are represented.*

We will note that usage of the **Run** function illustrates one of useful enough methods of providing the interface with the basic operational platform, but here*two* very essential moments take place. Above all, the function on some operational platforms*(for example***,***Windows XP Professional)* demands certain external reaction of the user at an exit from the**Mathematica** environment into an operational environment, and secondly, a call by means of the**Run** function of functions or the system***DOS*** commands assumes their existence in the directories system determined by the**$Path** variable since otherwise **Mathematica** doesn't recognize them. In particular, similar situation takes place in the case of usage of the external***DOS*** commands, for this reason in realization of the procedures**Attrib** and**Attrib1** that thru the**Run** function use the external***attrib*** command of***DOS*** system, a connection to system of directories of***$Path*** of the directories containing the***"attrib.exe"*** utility has been provided whereas for internal commands of the***DOS*** it isn't required.

So, at using of the internal ***dir*** command of***DOS*** system of an extension of the list of directories defined by the***$Path*** isn't required. At the same time, on the basis of standard*reception* on the basis of extension of the list defined by the***$Path*** variable the**Mathematica** doesn't recognize the external***DOS*** commands. In this regard a rather simple procedure has been created whose *successful* call**LoadExtProg[*x*]** provides search in file system of the computer of a program***x*** given by the full name with its subsequent copying into the subdirectory defined by the call**Directory[].** The successful procedure call **LoadExtProg[*x*]** searches out a datafile***x*** in file system of the computer and copies it into the directory defined by the function call**Directory[],**returning **Directory[]<>"\"<>*x*** if the datafile already was in this subdirectory or has been copied into this directory. In addition the first directory containing the found datafile***x*** supplements the list of the directories determined by the predetermined***$Path*** variable. Whereas the procedure call**LoadExtProg[*x*, *y*]** with the second optional argument***y*** ***–****an undefinite variable****–*** in addition through***y*** returns the list of all full paths to the found datafile***x*** without a modification of the directories list determined by the predetermined***$Path*** variable. In the case of absence of opportunity to find a required datafile***x*** ***$Failed*** is returned. The following fragment represents source code of the **LoadExtProg** procedure with examples of its application, in particular, for uploading into the directory defined by the the function call**Directory[]** of a copy of external***"attrib.exe"*** command of***MS DOS*** with check of the result.

```
In[2566] := LoadExtProg[x_ /; StringQ[x], y___] := Module[{a = Directory[], b =
Unique["agn"], c, d, h}, If[PathToFileQ[x] && FileExistsQ[x], CopyFileToDir[x,
Directory[]], If[PathToFileQ[x] && ! FileExistsQ[x], $Failed, d = a <> "\" <> x;
If[FileExistsQ[d], d, h = FileExistsQ1[x, b];
If[h, CopyFileToDir[b[[1]], a];
If[{y}== {}, AppendTo[$Path,

FileNameJoin[FileNameSplit[b[[1]]][[1 ;;–2]]]], y = b]; d, $Failed]]]]]

In[2567] := LoadExtProg["C:\attrib.exe"]
Out[2567]= $Failed
In[2568]:= LoadExtProg["attrib.exe"]
Out[2568]= "C:\Users\Aladjev\Documents\attrib.exe" In[2569]:=
FileExistsQ[Directory[] <> "\" <> "attrib.exe"]
```

Out[2569]= True
In[2570]**:= LoadExtProg["tlist.exe"]**
Out[2570]= $Failed
In[2571]**:= LoadExtProg["tasklist.exe", t]**
Out[2571]= **"**C**:**\Users\Aladjev\Documents\tasklist.exe**"** In[2572]**:= t**
Out[2572]= {**"**C**:**\WINDOWS\System32\tasklist.exe**",**

" C**:**\WINDOWS\SysWOW64\tasklist.exe**",**
"C**:**\WINDOWS\winsxs\amd64_microsoft\–windows–
tasklist_31bf3856ad364e35_6.1.7600.16385_none_ 843823d87402ab36\tasklist.exe**",**
"C**:**\WINDOWS\winsxs\x86_microsoft–windows–
tasklist_31bf3856ad364\e35_6.1.7600.16385_none_ 28198854bba53a00\tasklist.exe**"**}
In[2573]**:= Attrib1["C:\Temp\Cinema.txt", "Attr"]**
Out[2573]= {**"**A**"**, **"**S**"**, **"**R**"**}

Therefore, in advance by means of the call **LoadExtProg[*x*]** it is possible to provide access to a necessary datafile***x*** if, of course, it exists in file system of the computer. Thus, using the**LoadExtProg** procedure in combination with the system**Run** function, it is possible to carry out a number of very useful {***exe|com***}**-**programs in the environment of the**Mathematica–** the programs of different purpose which are absent in file system of**Mathematica** thereby significantly extending the functionality of the software of the**Mathematica** system that can be quite demanded by wide range of various appendices.

The above **LoadExtProg** procedure along with our**FileExistsQ1** procedure also uses the**CopyFileToDir** procedure whose the call**CopyFileToDir[*x*,*y*]** provides copying of a datafile or directory***x*** into a directory***y*** with return of the full path to the copied datafile or directory. If the copied datafile already exists, it isn**'**t updated if the target directory already exists, the directory***x*** is copied into its subdirectory of the same name. The next fragment represents source code of the**CopyFileToDir** procedure with examples of its usage.

In[2557]**:= CopyFileToDir[x_ /; PathToFileQ[x], y_ /; DirQ[y]] := Module[**{**a, b**}**,**
If[DirQ[x], CopyDir[x, y], If[FileExistsQ[x], a = FileNameSplit[x][[–1]];
If[FileExistsQ[b = y <> "\" <> a], b, CopyFile[x, b]], $Failed]]]

In[2558] **:= CopyFileToDir["C:\Temp\Cinema.txt", "C:\Mathematica"]** Out[2558]= **"**C**:**\Mathematica\Cinema.txt**"**
In[2559]**:= CopyFileToDir["C:\Temp", "C:\Mathematica\Temp"]** Out[2559]= **"**C**:**\Mathematica\Temp\Temp**"**
In[2560]**:= CopyFileToDir["C:\Temp\Gefal.htm", "C:\Mathematica"]** Out[2560]= **"**C**:**\Mathematica\Gefal.htm**"**

The given procedure has a variety of appendices in problems of processing of file system of the computer.

In conclusion of the section a rather useful procedure is represented which provides only two functions**–** ***(1)*** obtaining the list of the attributes ascribed to a datafile or directory, and***(2)*** removal of all ascribed attributes. The call **Attribs[*x*]** returns the list of attributes in string format which are ascribed to a datafile or directory***x*.** On the main directories of volumes of direct access the procedure call**Attribs** returns**$Failed.** While the call**Attribs[*x*, *y*]** with the second optional argument***y*** *–an expression–* deletes all attributes which are ascribed to a datafile or directory***x*** with returning at a successful call***0.*** The following fragment represents source code of the procedure along with the most typical examples of its usage.

In[2660]:= **Attribs[x_ /; FileExistsQ[x] || DirectoryQ[x], y___] := Module[{b, a = StandPath[x], c =“attrib.exe”, d =ToString[Unique[“g”]], g},**

If[DirQ[x] && StringLength[x] == 3 && StringTake[x, {2, 2}] == “:”, $Failed, g[] := Quiet[DeleteFile[Directory[] <>”\” <> c]]; If[! FileExistsQ[c], LoadExtProg[c]];

If[{y}== {}, Run[c <> ” ” <> a <> ” > “, d]; g[]; b = Characters[StringReplace[StringTake[Read[d, String], {1,–StringLength[a]–1}], ” “–> ””]]; DeleteFile[Close[d]]; b, a = Run[c <> “–A–H–R–S ” <> a]; g[]; a]]]

In[2661] **:= Attribs[“C:\Temp\Avz”]**
Out[2661]= {”A“, “S“, “H“, “R”}
In[2662]**:= Map[Attribs, {“C:/“, “E:\”}]**
Out[2662]= {$Failed**,** $Failed}
In[2663]**:= Attribs[“C:/Temp/Agn/aaa bbb ccc”]**
Out[2663]= {”A“, “R”}
In[2664]**:= Attribs[“C:/Temp/Agn/Elisa.pdf”]**
Out[2664]= {”R”}
In[2665]**:= Attribs[“C:/Temp/Agn/Vsv\G r s u.doc”]**
Out[2665]= {”A“, “R”}
In[2666]**:= Attribs[“C:/Temp/Agn/Vsv\G r s u.doc”, 90]**
Out[2666]= 0
In[2667]**:= Attribs[“C:/Temp/Agn/Vsv\G r s u.doc”]**
Out[2667]= {}

It should be noted that as argument ***x*** the usage of an existing datafile, full path to a datafile, or a directory is supposed. At that, the file***“attrib.exe”*** is removed from the directory defined by the call**Directory[]** after a call of the procedure**.** The**Attribs** procedure is enough fast-acting, supplementing the procedures**Attrib** and**Attrib1.** The**Attribs** procedure is effectively applied in programming of certain means of access to elements of file system of the computer at processing their attributes. Thus, it should be noted once again that the**Mathematica** has no standard means for processing of attributes of datafiles and directories therefore the offered procedures**Attrib, Attrib1** and **Attribs** in a certain measure fill this niche.

So, the declared possibility of extension of the system of directories which is defined by the***$Path*** variable, generally doesn**’**t operate already concerning the external***DOS*** commands what well illustrates both consideration of the above our procedures**Attrib,Attrib1,LoadExtProg** and an example with the external***“tlist”*** command that is provided display of all active processes of the current session with*WindowsXPProfessional* system, namely**:**

In[2565] **:= Run[“tlist”, ” > “, “C:\Temp\tlist.txt”]**
Out[2565]= 1
In[2566]**:= LoadExtProg[“tlist.exe”];**

Run["tlist", " > ", "C:\Temp\tlist.txt"] Out[2566]= 0

```
0 System Process 4 System
488 smss.exe

520 avgchsvx.exe
676 csrss.exe
716 winlogon.exe
760 services.exe
772 lsass.exe
940 ati2evxx.exe
960 svchost.exe
1016 svchost.exe
1092 svchost.exe
1240 svchost.exe
1300 vsmon.exe
1368 ati2evxx.exe 1656 explorer.exe 1680 ctfmon.exe
212 spoolsv.exe
348 svchost.exe
392 avgwdsvc.exe
660 jqs.exe
1168 svchost.exe
1448 MsPMSPSv.exe Program Manager 1548 AVGIDSAgent.exe
2204 avgnsx.exe
2284 avgemcx.exe
2852 alg.exe
3600 zlclient.exe
3764 avgtray.exe
3884 vprot.exe
3936 Skype.exe
4056 AVGIDSMonitor.exe 3316 AmplusnetPrivacyTools.exe 2256
FreeCommander.exe – 2248 WINWORD.EXE
4348 avgrsx.exe
4380 avgcsrvx.exe
5248 Mathematica.exe
FreeCommander
Mathematica_Book–Microsoft Word

Wolfram Mathematica 10.0 – [Running.AVZ_Package.nb] 4760 MathKernel.exe
4080 javaw.exe
4780 cmd.exe
4808 tlist.exe
C:\WINDOWS\system32\cmd.exe
```

The first example of the previous fragment illustrates, that the attempt by means of the**Run** function to execute the external***tlist*** command of***DOS*** completes unsuccessfully*(return code**1**)* whereas a result of the procedure call**LoadExtProg["*tlist.exe*"]** with search and download into the directory defined by the call**Directory[]** of the***"tlist.exe"*** file, allows to successfully execute by means of the**Run** the external command tlist with preservation of result of its performance in the***txt***file whose context is presented in the text by the shaded area.

Meanwhile, use of external software on the basis of the **Run** function along with possibility of extension of functionality of the**Mathematica** causes a rather serious*portability* question. So, the means developed by means of this technique with use the external***DOS*** commands are subject to influence of variability of the***DOS*** commands depending on version of a basic operating system. In a number of cases it demands a certain adaptation of the software according to a basic operating system.

7.4. Additional means of processing of datafiles and directories of file system of the computer

This section represents means of processing of datafiles and directories of file system of the computer that supplement and in certain cases and extend means of the previous section. Unlike the system functions**DeleteDirectory** and**DeleteFile** the following**DelDirFile** procedure removes a directory or datafile***x*** from file system of the computer, returning*Null*, i.e. nothing. At that, the procedure call**DelDirFile[*x*]** with one argument***x*** is analogous to a call**DeleteFile[*x*]** or**DeleteDirectory[*x*]** depending on the type of argument ***x*** **–** a datafile or directory. Whereas the call**DelDirFile[*x, y*]** with the second optional argument***y*** **–***an arbitrary expression***–** deletes a datafile or a catalog even if datafile***x*** or elements of directory***x*** of file system of the computer have the***Read-only*** attribute**;** in that case before its removal the attributes of an element***x*** are cancelled, providing correct removal of the element***x*** what unlike the system means expands opportunities for removal of elements of file system of the computer. The procedure eccentially uses our procedures **Attribs, DirQ, StandPath.** The following fragment represents source code of the**DelDirFile** procedure along with examples of its most typical usage.

```
In[2560]:=DelDirFile[x_ /; StringQ[x] && DirQ[x]||FileExistsQ[x], y___]:=
Module[{c, f, a = {}, b = "", k = 1}, If[DirQ[x] && If[StringLength[x] == 3 &&
StringTake[x, {2, 2}] == ":", False, True],

If[ {y}=={}, Quiet[DeleteDirectory[x, DeleteContents–> True]], a = {}; b = ""; c =
StandPath[x]; f = "$Art2618Kr$"; Run["Dir " <> c <> " /A/B/OG/S > " <> f];
Attribs[c, 90]; For[k, k < Infinity, k++, b = Read[f, String]; If[SameQ[b, EndOfFile],
DeleteFile[Close[f]]; Break[], Attribs[b, 90]]]; DeleteDirectory[x, DeleteContents–>
True]], If[FileExistsQ[x], If[{y}!={}, Attribs[x, 90]]; Quiet[DeleteFile[x]], $Failed]]]

In[2561]:= DelDirFile["F:\"]
Out[2561]= $Failed

In[2562] := DeleteFile["C:\Temp\Excel11.pip"]
DeleteFile::privv: Privilege violation during DeleteFile… >>
Out[2562]= $Failed
In[2563]:= DelDirFile["C:\Temp\Excel11.pip", 90]
In[2564]:= FileExistsQ["C:\Temp\Excel11.pip"]
Out[2564]= False
```

```
In[2565]:= Map1[{DirectoryQ, Attribs}, {"C:\Temp\Agn"}]
Out[2565]= {True, {"A", "S", "H", "R"}}
In[2566]:= DelDirFile["C:\Temp\Agn"]
Out[2566]= $Failed
In[2567]:= DelDirFile["C:\Temp\Agn", 500]
In[2568]:= DirectoryQ["C:\Temp\Agn"]
Out[2568]= False
```

Meanwhile, before representation of the following means it is expedient to determine one rather useful procedure whose essence is as follows. As it was already noted above, a file qualifier depends both on a register of symbols, and the used dividers of directories. Thus, the same datafile with different qualifiers**"C:\Temp\agn\cinema.txt"** and**"C:/temp\agn/cinema.txt"** opens in*two* various streams. Therefore its closing by means of the standard **Close** function doesn't close the***"cinema.txt"*** datafile, demanding closing of all streams on which it was earlier open. For solution of the given problem the**Close1** procedure presented by the next fragment has been determined.

```
In[2580] := Streams[]
Out[2580]= {OutputStream["stdout", 1], OutputStream["stderr", 2]} In[2581]:=
Read["C:/Temp\cinema.txt"]; Read["C:/Temp/Cinema.txt"];

Read["C: /Temp\cinema.txt"]; Read["c:/temp/birthday.doc"];
Read["C:/temp\BirthDay.doc"];
In[2582]:= Streams[]
Out[2582]= {OutputStream["stdout", 1], OutputStream["stderr", 2],
InputStream["C:/Temp\cinema.txt", 1697],
InputStream["C:/Temp/Cinema.txt", 1700],
InputStream["c:/temp/birthday.doc", 1705],
InputStream["C:/temp\BirthDay.doc", 1706]}
In[2583]:= Close["C:/Temp\cinema.txt"]
Out[2583]= "C:/Temp\cinema.txt"

In[2584] := Streams[]
Out[2584]= {OutputStream["stdout", 1], OutputStream["stderr", 2],
InputStream["C:/Temp/Cinema.txt", 1700],
InputStream["c:/temp/birthday.doc", 1705],
InputStream["C:/temp\BirthDay.doc", 1706]}

In[2585]:= Close1[x___String] := Module[{a = Streams[][[3 ;;–1]], b = {x}, c = {}, k =
1, j},

If[a == {}|| b == {}, {}, b = Select[{x}, FileExistsQ[#] &]; While[k <= Length[a], j =
1; While[j <= Length[b], If[ToUpperCase[StringReplace[a[[k]][[1]], {"\"–> "", "/"–>
""}]] == ToUpperCase[StringReplace[b[[j]], {"\"–> "", "/"–> ""}]], AppendTo[c,
a[[k]]]]; j++]; k++]; Map[Close, c]; If[Length[b] == 1, b[[1]], b]]]

In[2586] := Close1["C:/Temp\cinema.txt", "C:/temp\BirthDay.doc"] Out[2586]=
{"C:/Temp\cinema.txt", "C:/temp\BirthDay.doc"} In[2587]:= Streams[]
Out[2587]= {OutputStream["stdout", 1], OutputStream["stderr", 2]} In[2588]:= Close1[]
Out[2588]= {}
```

In[2589]:= **Close1["C:/Temp\cinema.txt", "C:/temp\BirthDay.doc"]** Out[2589]= {}
In[2590]:= **Close1["C:/Temp\Agn/Cinema.txt", AvzAgnVsvArtKr]** Out[2590]= Close1["C:/Temp\Agn/Cinema.txt", AvzAgnVsvArtKr] In[2591]:= **Closes[x_] := Quiet[Check[Close[x], Null]]**
In[2591]:= **Closes["C:\Temp\Svetlana\Kherson\Cinema.txt"]**

In[2667] := **Close2[x___String] := Module[{a = Streams[][[3 ;;–1]], b = {}, c, d = Select[{x}, StringQ[#] &]}, If[d == {}, {}, c[y_] := ToLowerCase[StringReplace[y, "/"–> "\"]]; Map[AppendTo[b, Part[#, 1]] &, a];**

d = DeleteDuplicates[Map[c[#] &, d]];

Map[Close, Select[b, MemberQ[d, c[#]] &]]]] In[2668]:= **Close2[]**
Out[2668]= {}

In[2669]:= **Close2["C:/Temp\cinema.txt", "C:/temp\BirthDay.doc"]** Out[2669]= {"C:/Temp\cinema.txt", "C:/Temp/Cinema.txt",

" c:/temp/birthday.doc", "C:/temp\BirthDay.doc"} In[2670]:= **Streams[]**
Out[2670]= {OutputStream["stdout", 1], OutputStream["stderr", 2]}

The procedure call **Close1[*x, y, z, …*]** closes all off really–existing datafiles in a list {***x, y, z, …***} irrespective of quantity of streams on which they have been opened by various files qualifiers with returning their list. In other cases the call on admissible actual arguments returns the empty list, i.e. {} whereas on inadmissible actual arguments a call is returned unevaluated. The previous fragment represents source code of the**Close1** procedure with examples of its usage. In end of the fragment the simple**Closes** function and the**Close2** procedure are presented. The function call**Closes[*x*]** returns nothing, closing a datafile***x,*** including the closed and empty datafiles without output of any erroneous messages. In certain appendices this function is quite useful. The procedure**Close2** is a functional analog of the above procedure**Close1.** The procedure call**Close2[*x,y, z, …*]** closes all off really–existing datafiles in a list {***x,y, z, …***} irrespective of quantity of streams on which they have been opened by various files qualifiers with returning their list. In other cases the call on admissible actual arguments returns the empty list, i.e. {} whereas on inadmissible actual arguments a call is returned unevaluated. The previous fragment represents source code of the**Close2** procedure with examples of its usage. In a number of appendices**Close1** and**Close2** are quite useful.

The following **DelDirFile1** procedure– an useful enough extension of the **DelDirFile** procedure on case of open datafiles in addition to the***Read-only*** attribute of both the separate datafiles, and the datafiles being in the deleted directory. The call**DelDirFile1[*x*]** is equivalent to the call**DelDirFile[*x, y*],** providing removal of a datafile or directory***x*** irrespective of openness of a separate datafile***x*** and the***Read-only*** attribute ascribed to it, or existence of similar datafiles in a directory***x.***The fragment below represents source code of the**DelDirFile1** procedure along with typical examples of its usage.

In[2725] := **DelDirFile1[x_ /; StringQ[x] && FileExistsQ[x] || DirQ[x] && If[StringLength[x]== 3 && StringTake[x, {2, 2}] == ":", False, True]] := Module[{a = {}, b = "", c = StandPath[x], d, f = "$Art590Kr$", k = 1}, If[DirQ[x], Run["Dir " <> c <> " /A/B/OG/S > "<>f]; Attribs[c, 90]; For[k, k < Infinity, k++, b = Read[f, String]; If[SameQ[b, EndOfFile], DeleteFile[Close[f]]; Break[],**

Attribs[b, 90]; Close2[b]]]; DeleteDirectory[x, DeleteContents–> True], Close2[x]; Attribs[x, 90]; DeleteFile[x]]]

In[2726] **:= Map[Attribs, {"C:/Temp\Agn/Cinema.txt", "C:/Temp\Agn/BirthDay.doc", "C:/Temp\Agn"}]**
Out[2726]= {{"A", "S", "H", "R"}, {"A", "S", "H", "R"}, {"A", "S", "H", "R"}}
In[2727]**:= Read["C:/Temp\Agn/Cinema.txt"]; Read["C:/Temp\Agn/BirthDay.doc"];**
In[2728]**:= Streams[]**
Out[2728]= {OutputStream["stdout", 1], OutputStream["stderr", 2], InputStream["C:/Temp\Agn/Cinema.txt", 131], InputStream["C:/Temp\Agn/BirthDay.doc", 132]}
In[2729]**:= DelDirFile1["C:/Temp\Agn"]**
In[2730]**:= Streams[]**
Out[2730]= {OutputStream["stdout", 1], OutputStream["stderr", 2]}
In[2731]**:= DirQ["C:\Temp\Agn"]**
Out[2731]= False
In[2732]**:= Attribs["C:\GrGu_Books\Cinema.TXT"]**
Out[2732]= {"A", "S", "H", "R"}
In[2733]**:= Read["C:\GrGu_Books\cinema.TXT"];**
In[2734]**:= Streams[]**
Out[2734]= {OutputStream["stdout", 1], OutputStream["stderr", 2], InputStream["C:\GrGu_Books\cinema.TXT", 149]}
In[2735]**:= DelDirFile1["C:\GrGu_Books\cinema.TXT"]**
In[2736]**:= FileExistsQ["C:\GrGu_Books\cinema.TXT"]**
Out[2736]= False

The means representing quite certain interest at working with file system of the computer as independently, and as a part of means of processing of the datafiles and directories complete this section. They are used and by a series of means of our***AVZ_Package*** package [48]. In particular, at working with files the**OpenFiles** procedure can be rather useful, whose call**OpenFiles[]** returns the***2***–element nested list**,** whose the first sublist with the first***"read"*** element contains full paths to the datafiles opened on reading whereas the second sublist with the first***"write"*** element contains full paths to the files opened on writing in the current session. In the absence of such datafiles the procedure call returns the empty list, i.e. {}. Whereas the call**OpenFiles[*x*]** with one actual argument***x*** *–a datafile classifier–* returns result of the above format relative to the open datafile***x*** irrespective of a format of coding of its qualifier. If***x*** defines a closed or nonexistent datafile then the procedure call returns the empty list, i.e. {}. The fragment below represents source code of the procedure along with rather typical examples of its use.

In[2628] **:= OpenFiles[x___String] := Module[{a = Streams[][[3 ;;–1]], b, c, d, h1 = {"read"}, h2 = {"write"}}, If[a == {}, {}, d = Map[{Part[#, 0], Part[#, 1]}&, a]; b = Select[d, #[[1]] == InputStream &];**

c = Select[d, #[[1]] == OutputStream &]; b = Map[DeleteDuplicates,

Map[Flatten, Gather[Join[b, c], #1[[1]] == #2[[1]] &]]]; b = Map[Flatten, Map[If[SameQ[#[[1]], InputStream], AppendTo[h1, #[[2 ;;–1]]], AppendTo[h2, #[[2

;;–1]]]] &, b]]; If[{x}== {}, b, If[SameQ[FileExistsQ[x], True], c = Map[Flatten,

Map[{#[[1]], Select[#, StandPath[#] === StandPath[x] &]}&, b]]; If[c == {{"read"}, {"write"}}, {}, c = Select[c, Length[#] > 1 &]; If[Length[c] > 1, c, c[[1]]], {}]]]]]

In[2629] **:= OpenFiles[]**
Out[2629]= {{"read", "C:/Temp\cinema.txt", "C:/Temp/Cinema.txt", "c:/temp/birthday.doc", "C:/temp\BirthDay.doc", "C:/GrGu_Books/Birthday1.doc"}, {"write", "C:\GrGu_Books\Birthday1.doc"}}
In[2630]**:= OpenFiles["AvzArnVsvArtKr"]**
Out[2630]= {}
In[2631]**:= OpenFiles["C:\Temp\Cinema.txt"]**
Out[2631]= {"read", "C:/Temp\cinema.txt", "C:/Temp/Cinema.txt"}

In[2632]**:= OpenFiles["C:\GrGu_Books/Birthday1.doc"]**
Out[2632]= {{"read", "C:/GrGu_Books/Birthday1.doc"}, {"write", "C:\GrGu_Books\Birthday1.doc"}}

At that, as the full path it is understood or really full path to a datafile in file system of the computer, or its*full name* if it is located in the current directory determined by the function call**Directory[].**

As the procedure similar to the **OpenFiles,** the following procedure can be used, whose call**StreamFiles[]** returns the nested list from two sublists, the first sublist with the first***"in"*** element contains full paths/names of the files opened on the reading while the second sublist with the first***"out"*** element contains full paths/names of the datafiles opened on the recording. Whereas in the absence of the open datafiles the procedure call**StreamFiles[]** returns ***"AllFilesClosed".*** The next fragment presents source code of the**OpenFiles** procedure along with some typical examples of its usage.

In[2555] **:= StreamFiles[] := Module[{a = Map[ToString1, StreamsU[]], b ={}, w = {"out"}, r = {"in"}, c, k = 1}, If[a == {}, Return["AllFilesClosed"], For[k, k <= Length[a], k++, c = a[[k]]; If[SuffPref[c, "Out", 1], AppendTo[w, StrFromStr[c]], AppendTo[r, StrFromStr[c]]]]]]; c = Select[Map[Flatten, {r, w}], Length[#] > 1 &]; If[Length[c] == 1, c[[1]], c]]**

In[2556]**:= StreamFiles[]**
Out[2556]= {{"in", "C:/Temp\cinema.txt", "C:/Temp/Cinema.txt",

" c:/temp/birthday.doc", "C:/temp\BirthDay.doc"}, {"out", "C:\GrGu_Books\Birthday1.doc"}}
In[2557]**:= Close["C:\GrGu_Books\Bithday1.doc"]**
Out[2557]= "C:\GrGu_Books\Birthday1.doc"
In[2558]**:= StreamFiles[]**
Out[2558]= {"in", "C:/Temp\cinema.txt", "C:/Temp/Cinema.txt",

" c:/temp/birthday.doc", "C:/temp\BirthDay.doc"} In[2559]**:= CloseAll[]; StreamFiles[]**
Out[2559]= "AllFilesClosed"

In[2560] **:= Read["Book_3.doc"];**
In[2561]**:= StreamFiles[]**
Out[2561]= {"in", "Book_3.doc"}

In a number of cases at work with datafiles the following procedure can be very useful, whose call**IsFileOpen[*f*]** returns*True* if a datafile***f*** determined by a name or full path is open and*False* otherwise. If the argument***f*** doesn't define an existing datafile the procedure call is returned unevaluated. While the call**IsFileOpen[*f*,*h*]** with the second optional argument***h*** *–an undefinite variable–* returns through***h*** the nested list whose elements are sublists of the format {{***"read"***|***"write"***}**,** {*The list of streams on which the datafilefis open on reading|recording*}} if the main result is*True***.** The fragment below represents source code of the**IsFileOpen** procedure along with examples of its usage.

```
In[2550]:= IsFileOpen[F_ /; FileExistsQ[Ff], h___] :=
Module[{a = OpenFiles[F]}, If[a == {}, False, If[{h}!= {}&& ! HowAct[h], h = a,
Null]; True]]

In[2551] := OpenWrite["C:/temp/cinema.doc"];
OpenRead["C:/temp\cinema.doc"];
In[2552]:= Streams[]
Out[2552]= {OutputStream["stdout", 1], OutputStream["stderr", 2],
OutputStream["C:/temp/cinema.doc", 84],
InputStream["C:/temp\cinema.doc", 85]}
In[2553]:= IsFileOpen["C:/Temp\Cinema.doc", t90]
Out[2553]= True
In[2554]:= t90
Out[2554]= {{"read", {"C:/temp/cinema.doc"}},
{"write", {"C:/temp\cinema.doc"}}}
In[2555]:= Read["C:/temp/birthday.doc"];
In[2556]:= IsFileOpen["C:\temp\BirthDay.doc", h500]
Out[2556]= True
In[2557]:= h500
Out[2557]= {"read", {"C:/temp/birthday.doc"}}
In[2558]:= CloseAll[]; IsFileOpen["C:\temp\BirthDay.doc"]
Out[2558]= False
```

The following quite simple procedure is represented as a rather useful tool at operating with file system of the computer, whose the call**DirEmptyQ[*d*]** returns*True* if a directory***d*** is empty, otherwise*False* is returned. Moreover, the call**DirEmptyQ[*d*]** is returned unevaluated if***d*** isn't a*real* directory. The following fragment presents source code of the**DirEmptyQ** procedure with typical enough examples of its usage.

```
In[2602]:= DirEmptyQ[d_ /; DirQ[d]] := Module[{a = "$DirFile$", b, c, p =
StandPath[StringReplace[d, "/"–> "\"]], h = " 0 File(s) "}, b = Run["Dir " <> p <>
If[SuffPref[p, "\", 2], "", "\"] <> "*.* > " <> a]; If[b != 0, $Failed, Do[c = Read[a,
String], {6}]]; DeleteFile[Close[a]]; ! StringFreeQ[c, h]]

In[2603] := Map[DirEmptyQ, {"C:\Mathematica/Avz", "C:/temp", "C:\",
"c:/Mathematica", "Rans", "c:/Mathematica/Avz/Agn/Art/Kr"}]
Out[2603]= {False, False, False, False, DirEmptyQ["Rans"], True}
In[2604]:= DirEmptyQ["C:\Mathematica/Avz/Agn/Art/Kr/"]
Out[2604]= True
In[2605]:= Map[DirEmptyQ, {"C:\Mathematica/Avz", "C:/Temp/", "C:/"}]
```

Out[2605]= {False**,** False**,** False}
In[2606]**:= DirEmptyQ["C:\Program Files (x86)"]**
Out[2606]= False

At that in addition to the previous**DirEmptyQ** procedure the procedure call **DirFD[*j*]** returns the two-element nested list whose the first element defines the list of subdirectories of the first nesting level of a directory***j*** whereas the second element**–** the list of datafiles of a directory***j;*** if a directory***j*** is empty, the procedure call returns the empty list, i.e. {}. The fragment below presents source code of the**DirFD** procedure along with examples of its usage.

In[2575] **:= DirFD[d_ /; DirQ[d]] := Module[**{**a =”$DirFile$”, b =**{{},{}}**, c, h, t, p = StandPath[StringReplace[d, “/”–> “\”]]**}**, If[DirEmptyQ[p], Return[**{}**], Null]; c = Run["Dir ” <> p <> ” /B ” <> If[SuffPref[p, “\”, 2], ””, “\”] <> “*.* > ” <>a]; t =Map[ToString, ReadList[a, String]]; DeleteFile[a]; Map[**{**h = d <> “\” <> #; If[DirectoryQ[h], AppendTo[b[[1]], #], If[FileExistsQ[h], AppendTo[b[[2]], #], Null]]**}**&, t]; b]** In[2576]**:= DirFD["C:/Program Files/Wolfram Research/Mathematica/10.1 \Documentation\English\Packages”]**

Out[2576] = {{”ANOVA“**,** “Audio“**,** “AuthorTools“**,** “BarCharts“**,** “Benchmarking“**,** “BlackBodyRadiation“**,** “Calendar“**,** “Combinatorica“**,** “Compatibility“**,** “ComputationalGeometry“**,** “ComputerArithmetic“**,** “Developer“**,** “EquationTrekker“**,** “ErrorBarPlots“**,** “Experimental“**,** “FiniteFields“**,** “FourierSeries“**,** “FunctionApproximations“**,** “Geodesy“**,** “GraphUtilities“**,** “HierarchicalClustering“**,** “Histograms“**,** “HypothesisTesting“**,** “LinearRegression“**,** “MultivariateStatistics“**,** “Music“**,** “NonlinearRegression“**,** “Notation“**,** “NumericalCalculus“**,** “NumericalDifferentialEquationAnalysis“**,** “PhysicalConstants“**,** “PieCharts“**,** “PlotLegends“**,** “PolyhedronOperations“**,** “Polytopes“**,** “PrimalityProving“**,** “Quaternions“**,**
“RegressionCommon“**,** “ResonanceAbsorptionLines“**,** “Splines“**,** “StandardAtmosphere“**,** “StatisticalPlots“**,** “Units“**,** “VariationalMethods“**,** “VectorAnalysis“**,** “VectorFieldPlots“**,** “WorldPlot“**,** “XML”}**,** {}}

In[2577] **:= Length[%[[1]]]**
Out[2577]= 48
In[2578]**:= DirFD["C:\Program Files”]**
Out[2578]= {{”Common Files“**,** “Dell Inc“**,** “DVD Maker“**,** “Extras“**,**

“ File Association Helper“**,** “Intel“**,** “Internet Explorer“**,** “MSBuild“**,** “Nitro“**,** “Realtek“**,** “Reference Assemblies“**,** “Softland“**,** “Windows Defender“**,** “Windows Journal“**,** “Windows Mail“**,** “Windows Media Player“**,** “Windows NT“**,** “Windows Photo Viewer“**,** “Windows Portable Devices“**,** “Windows Sidebar“**,** “Wolfram Research“**,** “Intel”}**,** {}}

In[2579] **:= DirFD["C:\Temp”]**
Out[2579]= {{”Dialog_files”}**,** {”aaa.txt“**,** “Addresses_for_book.doc“**,** “Books.doc“**,** “Books.mht“**,** “Birthday.doc“**,** “cinema.doc“**,** “Cinema.txt“**,** “Dialog.htm“**,** “ISSN Application form.pdf“**,** “Math_Trials.DOC“**,** “potencial.txt“**,** “regcleaner.exe”}}

In particular, in an example of the previous fragment the list of directories with documentation on the packages delivered with the**Mathematica*10.1.0*** system is returned. Thus, with release***10.1.0*** of the**Mathematica** system***48*** packages of different purpose which is rather simply seen from the name of the subdirectories containing them are being delivered.

In addition to the**DirFD** procedure the**DirFull** procedure represents a quite certain interest, whose call**DirFull[*d*]** returns the list of all full paths to the subdirectories and files contained in a directory***d*** and its subdirectories***;*** the first element of this list**–** the directory***d.*** While on an empty directory***d*** the call**DirFull[*d*]** returns the empty list, i.e. {}. The fragment below represents source code of the**DirFull** procedure along with examples of its usage.

In[2595]**:= DirFull[x_ /; DirQ[x]] := Module[**{**a = "$Art26Kr18$", c, b = StandPath[StringReplace[x, "/"–> "\"]]**}**, If[DirEmptyQ[x],** {}**, Run["Dir /S/B/A ", b, " > ", a]; c = Map[ToString, ReadList[a, String]]; DeleteFile[a]; Prepend[c, b]]]**

In[2596] **:= DirFull["C:\Mathematica\avz\agn/Art/Kr"]**
Out[2596]= {}
In[2597]**:= DirFull["C:\Users\Aladjev\DownLoads"]**
Out[2597]= {**"**c**:**\users\aladjev\downloads**",**

" c**:**\users\aladjev\downloads\Book_Grodno.doc**",**
"c**:**\users\aladjev\downloads\CuteWriter.exe**", "**c**:**\users\aladjev\downloads\desktop.ini**",**
"c**:**\users\aladjev\downloads\IMG_0389.MOV**",**
"c**:**\users\aladjev\downloads\Mathematica_10.1.0_WIN.zip"}

In addition to the**DirFull** procedure the call**TypeFilesD[*d*]** of the procedure **TypeFilesD** returns the sorted list of types of the files located in a directory ***d*** with returning of***"undefined"*** on datafiles without of a name extension. At that, the datafiles located in the directory***d*** and in all its subdirectories of an arbitrary nesting level are considered. Moreover, on the empty directory***d*** the procedure call**TypeFilesD[*d*]** returns the empty list, i.e. {}. The following fragment represents source code of the**TypeFilesD** procedure along with typical enough examples of its usage.

In[5180] **:= TypeFilesD[x_ /; DirQ[x]] := Module[**{**a = "$Art26Kr18$", d =** {}**, c, p, b = StandPath[StringReplace[x, "/"–> "\"]]**}**, If[DirEmptyQ[x],** {}**, Run["Dir /S/B/A ", b, " > ", a]; c = Map[ToString, ReadList[a, String]]; DeleteFile[a];**

Sort[Select[DeleteDuplicates[Map[If[DirectoryQ[#], Null, If[FileExistsQ[#], p = ToLowerCase[ToString[FileExtension[#]]]; If[! SameQ[p, ""], p, "undefined"]], Null]&, c]], ! SameQ[#, Null] &]]]]

In[5181]**:= TypeFilesD["c:\temp\"]**
Out[5181]= {**"**txt**", "**doc**", "**mht**", "**htm**", "**pdf**", "**exe**", "**jpg**", "**js**", "**css**", "**png**",**

" gif**", "**php**", "**json**", "**undefined**", "**query**", "**xml"}
In[5182]**:= TypeFilesD["C:/Tallinn\Grodno/Kherson"]**
Out[5182]= {}
In[5183]**:= TypeFilesD["C:\Mathematica"]**
Out[5183]= {**"**css**", "**doc**", "**gif**", "**htm**", "**jpg**", "**js**", "**json**", "**pdf**", "**png**", "**tmp"}
In[5184]**:= TypeFilesD["C:\Program Files (x86)\Maple 11"]** Out[5184]= {**"**access**",**
"afm**", "**bfc**", "**cfg**", "**cpl**", "**csv**", "**dat**", "**del**", "**dll**",**

“ dtd**“**, **“**ent**“**, **“**err**“**, **“**exe**“**, **“**gif**“**, **“**hdb**“**, **“**html**“**, **“**ico**“**, **“**ind**“**, **“**ini**“**, **“**ja**“**, **“**jar**“**, **“**jpg**“**, **“**jsa**“**, **“**lib**“**, **“**lic**“**, **“**mat**“**, **“**mla**“**, **“**mod**“**, **“**mw**“**, **“**pf**“**, **“**policy**“**, **“**properties**“**, **“**security**“**, **“**src**“**, **“**template**“**, **“**ttf**“**, **“**txt**“**, **“**undefined**“**, **“**vec**“**, **“**wav**“**, **“**xls**“**, **“**xml**“**, **“**xsd**“**, **“**xsl**”**}

The **FindFile1** procedure serves as useful extension of the standard**FindFile** function, providing search of a datafile within file system of the computer. The procedure call**FindFile1[*x*]** returns a full path to the found datafile***x*,** or the list of full paths*(if datafilexis located in different directories of file system of the computer)***,** otherwise the call returns the empty list, i.e. {}. While the call **FindFile1[*x*, *y*]** with the*second* optional argument***y*** *–full path to a directory–* returns a full path to the found datafile***x*,** or the list of full paths located in the directory***y*** and its subdirectories. The fragment below represents source code of the**FindFile1** procedure along with typical examples of its usage.

In[2550]**:= FindFile1[x_ /; StringQ[x], y___] := Module[**{**c, d =** {}**, k = 1, a = If[**{**y**}**!=** {}**&& PathToFileQ[y],** {**y**}**, Map[# <> “:\” &, Adrive[]]], b = “\” <> ToLowerCase[x]**}**, For[k, k <= Length[a], k++,**

c = Map[ToLowerCase, Quiet[FileNames[“*”, a[[k]], Infinity]]]; d = Join[d, Select[c, SuffPref[#, b, 2] && FileExistsQ[#] &]]]; If[Length[d] == 1, d[[1]], d]]

In[2551]**:= FindFile1[“Letter_5_02_15.doc”]**

Out[2551] = {**”**C**:**\Temp\Letter_5_02_15**.**doc**“, “**F**:**\Letter_5_02_15**.**doc**”**} In[2552]**:=**
FindFile1[“Cinema.txt”, “C:\Temp”]
Out[2552]= **“**c**:**\temp\cinema.txt**”**
In[2553]**:= FindFile1[“Cinema.txt”]**
Out[2553]= {**”**C**:**\GrGU_Books\Cinema.txt**“, “**C**:**\Program Files\

Wolfram Research \Mathematica\10.1\Cinema.txt**“, “**C**:**\Program Files\Wolfram Research\Mathematica\10.1
\SystemFiles\Cinema.txt**“, “**C**:**\Temp\Cinema.txt**“, “**E**:**\CD_Book\Cinema.txt**”**}
In[2554]**:= FindFile1[“AvzAgnVsvArtKr”]**
Out[2554]= {}
In[2555]**:= t = TimeUsed[]; FindFile1[“Book_3.doc”]; TimeUsed[]–t** Out[2555]= 5**.**928
In[2556]**:= t = TimeUsed[]; FileExistsQ1[“Book_3.doc”]; TimeUsed[]–t** Out[2556]= 5**.**335

In particular, ***2*** last example of the previous fragment indicate, the**FindFile1** in many respects is functionally similar to the**FileExistsQ1** procedure but it in the*temporary* relation is somewhat less fast-acting in the same file system of the computer.

It is possible to give the **SearchDir** procedure as one more quite indicative example, whose call**SearchDir[*d*]** returns the list of all paths in file system of the computer which are completed by a subdirectory***d;*** in case of lack of such paths the procedure call**SearchDir[*d*]** returns the empty list, i.e. {}. In combination with the procedures**FindFile1** and**FileExistsQ1** the**SearchDir** procedure is useful at working with file system of the computer, as confirms their usage for the solution of tasks of similar type. The following fragment represents source code of the**SearchDir** procedure with examples of its use.

In[2595]**:= SearchDir[d_ /; StringQ[d]] := Module[**{**a = Adrive[], c, t =** {}**, p, b = “\”**

<>ToLowerCase[StringTrim[d, ("\"|"/") …]]<> "\", g = {}, k = 1, v},

For[k, k <= Length[a], k++, p = a[[k]]; c = Map[ToLowerCase, Quiet[FileNames["*", p <> ":\", Infinity]]]; Map[If[! StringFreeQ[#, b] || SuffPref[#, b, 2] && DirQ[#], AppendTo[t, #], Null] &, c]]; For[k =1, k <= Length[t], k++, p = t[[k]]<>"\"; a = StringPosition[p, b]; If[a == {}, Continue[], a = Map[#[[2]] &, a]; Map[If[DirectoryQ[v = StringTake[p, {1, #–1}]], AppendTo[g, v], Null] &, a]]]; DeleteDuplicates[g]]

```
In[2596] := SearchDir["AvzAgnVsvArtKr"]
Out[2596]= {}
In[2597]:= SearchDir["\Temp/"]
Out[2597]= {"c:\temp", "c:\users\aladjev\appdata\local\temp",

" c:\windows\assembly\nativeimages_v2.0.50727_32\temp",
"c:\windows\assembly\nativeimages_v2.0.50727_64\temp",
"c:\windows\assembly\nativeimages_v4.0.30319_32\temp",
"c:\windows\assembly\nativeimages_v4.0.30319_64\temp",
"c:\windows\assembly\temp", "c:\windows\temp",
"c:\windows\system32\driverstore\temp",
"c:\windows\winsxs\temp"}

In[2598] := SearchDir["Mathematica"]
Out[2598]= {"c:\mathematica", "c:\programdata\mathematica", "c:\program
files\wolfram research\mathematica", "c:\program files\wolfram
research\mathematica\10.1 \addons\packages\ guikit\src\mathematica",
"c:\users\aladjev\appdata\local\mathematica",
"c:\users\aladjev\appdata\roaming\mathematica", "c:\users\aladjev\mathematica",
"c:\users\all users\mathematica", "e:\mathematica"}
```

It is once again expedient to note that the mechanism of objects typification which the**Mathematica** system has, is a significantly inferior to the similar mechanism of the**Maple** system, but only relatively to the built-in types of testing of objects. Meanwhile, and means of the**Mathematica** system allow to test types of the most important objects. So, the system**FileType** function provides the checking be a directory or a datafile as illustrates the following simple enough examples, namely:

```
In[3742] := FileType["D:\Math_myLib"]
Out[3742]= Directory
In[3743]:= FileType["D:\Math_myLib\ArtKr.mx"]

Out[3743] = File
In[3744]:= FileExistsQ["D:\Math_myLib\ArtKr.mx"]
Out[3744]= True
In[3745]:= FileExistsQ["D:\Math_myLib"]
Out[3745]= True
```

In the mean time, these means yield to our procedures ***isFile*** and***isDir*** for the***Maple*** system, providing testing of datafiles and directories respectively [47]. Thus, the**isFile** procedure not only tests the existence of a datafile, but also the mode of its opening, what in certain cases is very important. There are other interesting enough means for testing of the state of directories and datafiles, including their types [25,47]. On the other hand, the**Mathematica** system posesses the**FileExistsQ** function that returns*True* if a tested object is a datafile or directory what from*standpoint* of file system of the computer is quite correctly while for the user working with datafiles it is not the same what rather visually illustrates the following very simple example, namely**:** In[2645]**:= F := "C:\Mathematica"; If[FileExistsQ[F], OpenRead[F];**

Read[F], Message[F::file, "file is absent"]] OpenRead**::**noopen**:** Cannot open C**:**\Mathematica.>> Read**::**openx**:** C**:**\Mathematica is not open**.** >>

Out[2645]= Read["D**:**\Mathematica"]

Check by means of the **FileExistsQ** function defines existence of the datafile ***F****(though instead of it the directory is specified)***,** then the attempt to open this datafile***F*** on the reading with the subsequent reading its first logical record are done, but both these procedures of access are completed with return of erroneous diagnostics. Therefore for this purpose it is necessary to use the testing function**IsFile** combining the functions**FileExistsQ** and**DirectoryQ** or somewhat more complex organized procedure whose call**FileQ[*f*]** returns *True* if the string***f*** defines a real–existing datafile, and*False* otherwise. The **FileQ** procedure serves sooner for a some illustration of development tools of the procedures oriented on working with file system of the computer. The fragment represents source codes of both means with examples of their use.

In[2622] **:= IsFile[x_] := If[FileExistsQ[x], If[! DirectoryQ[x], True, False], False]; Map[FileType, {"c:\mathem", "c:\mathem\ap.doc"}]**
Out[2622]= {Directory**,** File}

In[2623] **:= FileQ[f_ /; StringQ[f]] := Module[{d = Adrive[], s = {}, k = 1, a = ToLowerCase[StringReplace[Flatten[OpenFiles[]], "\\"–> "/"]], b = ToLowerCase[StringReplace[Directory[], "\"–> "/"]],**

c = ToLowerCase[StringReplace[f, "\" –> "/"]]},
For[k, k <= Length[d], k++, AppendTo[s, d[[k]] <> ":"]]; If[StringLength[c] < 2 ||

! MemberQ[ToLowerCase[s], StringTake[c, {1, 2}]], c = b <>"/" <>c, Null];
If[DirQ[c], False, If[MemberQ[a, c], True,
If[Quiet[OpenRead[c]] === $Failed, False, Close[c]; True]]]]

In[2624]**:= Map[FileQ, {"c:/Temp/Cinema.txt", "Book_3.doc", "E:/Art.Kr"}]**
Out[2624]= {True**,** True**,** False}
For the differentiated testing of files the**FileType** function is used too**:**

In[2552] **:= Map[FileType, {"c:/Mathematica", "c:/Mathematica/ap.doc"}]**
Out[2552]= {Directory**,** File}
The**Mathematica** system has also some other similar testing means oriented on processing of elements of file system of the computer. A number of such functions has been considered slightly above along with our means. So, the following fragment represents procedure**,** whose call**EmptyFileQ[*f*]** returns *True* if a datafile***f*** is empty, and*False* otherwise.

In[2640] **:= EmptyFileQ[f_ /; StringQ[f],y___] := Module[{a, b, c, d ={}, k =1}, If[FileExistsQ[f], b = {f}, c = Art26Kr18; ClearAll[Art26Kr18]; a = FileExistsQ1[f,**

Art26Kr18]]; If[! a, Return[$Failed], b = Art26Kr18; Art26Kr18 = c]; While[k <= Length[b], AppendTo[d, Quiet[Close[b[[k]]]]; If[Quiet[Read[b[[k]]]] === EndOfFile, Quiet[Close[b[[k]]]; True], Quiet[Close[b[[k]]]]; False]]; k++]; d =If[Length[d]==1, d[[1]], d]; If[{y}!= {}, {d, If[Length[b] == 1, b[[1]], b]}, d]]

```
In[2641] := Map[EmptyFileQ, {"c:/temp/cinema.txt", "c:/temp/cinema.doc"}]
Out[2641]= {False, True}
In[2642]:= EmptyFileQ["cinema.txt"]
Out[2642]= {False, True, False, False, False, True, False}
```

```
In[2643] := EmptyFileQ["C:\Cinema.txt", 90]
Out[2643]= {{False, True, False, False, False, True, False},
"C:\GrGU_Books\Cinema.txt", "C:\Mathematica\Cinema.txt", "C:\Program
Files\Wolfram Research\Mathematica\10.1\ Cinema.txt", "C:\Program Files\Wolfram
Research\ Mathematica\10.1\SystemFiles\Links\Cinema.txt", "C:\Temp\Cinema.txt",
"E:\Cinema.txt",
"E:\CD_Book\Cinema.txt"}}
In[2644]:= EmptyFileQ["Appendix.doc", 90]
Out[2644]= $Failed
In[2645]:= EmptyFileQ["E:\Cinema.txt", 500]
Out[2645]= {True, "E:\Cinema.txt"}
```

If a datafile ***f*** is absent in file system of the computer, the call**EmptyFileQ[*f*]** returns the**$Failed.** Moreover, if in the course of search of the datafile***f*** its multiplicity in file system of the computer is detected, all datafiles from list of the found datafiles are tested, including also datafiles that are located in the***Recycle Bin*** directory. At that, the procedure call**EmptyFileQ[*f*,*y*]** with two actual arguments where optional argument***y*** **–** an expression, returns the nested***2***–element list whose first sublist defines*emptiness*/*nonemptiness (True*|*False)* of the datafile***f*** in the list of datafiles of the same name whereas the second sublist defines full paths to the datafiles***f*** of the same name. At that, between both sublists the one–to–one correspondence takes place. The previous fragment represents both source code, and the typical examples of usage of the**EmptyFileQ** procedure.

The **FindSubDir** procedure provides search of the full paths that contain a subdirectory***x*** given by a full name in file system of the computer or in file system of the given devices of direct access that are determined by names in string format. The procedure call**FindSubDir[*x*]** returns the list of*full paths* within all file system of the computer, while the call**FindSubDir[*x*, *y*, *z*,…]–** within only file system of the devices {***y*, *z*,…**}. The next fragment represents source code of the**FindSubDir** procedure with examples of its application.

In[2542] **:= FindSubDir[x_ /; StringQ[x], y___] := Module[{b = {}, c = "", p, t, k = 1, a = If[{y}== {}, Adrive[], {y}], f = "Art26Kr18.txt", h = ToLowerCase[x]}, While[k <= Length[a], Run["Dir ", a[[k]] <> ":\", " /B/S/L > "<>f]; While[! SameQ[c, "EndOfFile"], c = ToString[Read[f, String]];**

t = FileNameSplit[c]; p = Flatten[Position[t, h]]; If[p !={}&& DirectoryQ[FileNameJoin[t[[1;; p[[1]]]]]], AppendTo[b, c]]; Continue[]]; Closes[f]; c

= ""; k++]; {DeleteFile[f], b}[[2]]]

In[2543] **:= FindSubDir["Dell Inc"]**
Out[2543]= {"c:\program files\dell inc", "c:\program files\dell inc\ dell edoc viewer", "c:\program files\dell inc\dell edoc viewer\ eddy.ini", "c:\program files\dell inc\dell edoc viewer\ edocs.exe", "c:\program files\dell inc\dell edoc viewer\ helppaneproxy.dll", "c:\program files\dell inc\dell edoc viewer\interop.helppane.dll", "c:\program files\dell inc\dell edoc viewer\sweepdocs.exe"}
In[2544]**:= FindSubDir["Dell Inc", "F"]**
Out[2544]= {}
In[2545]**:= FindSubDir["AVZ_Package", "C", "E"]**
Out[2545]= {"e:\avz_package", "e:\avz_package\avz_package.cdf",
"e:\avz_package\avz_package.m",
"e:\avz_package\avz_package.mx",
"e:\avz_package\avz_package.nb"}

The following **FilesDistrDirs** procedure in a certain degree bears structural character for a directory given by the actual argument of the procedure. The call**FilesDistrDirs[*x*]** returns the nested list whose elements– sublists of the following format {***dir_p, f1, f2, f3,…, fn***}, where***dir_p* –** a directory***x*** and all its subdirectories of any nesting level, whereas***f1, f2, f3, …, fn* –** names of the datafiles located in this directory. The following fragment represents source code of the**FilesDistrDirs** procedure along with an example of its usage.

In[2555] **:= FilesDistrDirs[x_ /; DirQ[x]] := Module[{a = {}, b, d, g, h = {}, t, c = FromCharacterCode[17], f = "$Art26Kr18$", k = 1}, Run["Dir " <> StandPath[x] <> " /A/B/OG/S > " <> f];**

For[k, k < Infinity, k++, b = Read[f, String]; If[SameQ[b, EndOfFile], DeleteFile[Close[f]]; Break[], AppendTo[a, b]]]; b = Gather[PrependTo[a, StringReplace[x, "/"–> "\"]], DirQ[#1] === DirQ[#2] &]; d = {Sort[Map[StringJoin[#, "\"] &, b[[1]]], StringCount[#1, "\"] >= StringCount[#2, "\"] &], Quiet[Check[b[[2]], {}]]}; a = Map[ToLowerCase, Flatten[d]]; For[k = 1, k <= Length[d[[1]]], k++, t = ToLowerCase[d[[1]][[k]]]; AppendTo[h, g = Select[a, SuffPref[#, t, 1] && StringFreeQ[StrDelEnds[#, t, 1], "\"] &]]; a = MinusList[a, g]]; a = {}; For[k = 1, k <= Length[h], k++, b = h[[k]]; AppendTo[a, {b[[1]], Map[StrDelEnds[#, b[[1]], 1] &, b[[2 ;;–1]]]}]]; Map[Flatten[#] &, a]]

In[2556]**:= FilesDistrDirs["C:\GrGU_Books"]**
Out[2556]= {{"c:\grgu_books\avz_package\", "avz_package.m",

" avz_package.mx", "avz_package.nb", "avz_package.cdf"}, {"c:\grgu_books\", "birthday.doc", "cinema.txt", "general_statistics.pdf", "general_statistics_cover.pdf", "iton14_5.pdf", "school.pdf"}}

The rather simple **PathToFileQ** function is useful at working with files and directories, whose call**PathToFileQ[*x*]** returns*True* if***x*** defines a potentially admissible full path to a directory or datafile, and*False* otherwise. The next fragment represents source code of the function with an example of its use.

In[2555]**:= PathToFileQ[x_ /; StringQ[x]] := If[StringLength[x] >= 3, If[MemberQ[Join[CharacterRange["a", "z"], CharacterRange["A", "Z"]],**

StringTake[x, 1]] && StringTake[x, {2, 2}] == ":" && And[Map3[StringFreeQ, x, {"/", "\"}]] !={True, True}, True, False], False]

In[2556] **:= Map[PathToFileQ, {"C:", "C:/", "G:/AVZ_Package", "H:\agn", "C:/Temp", "C:/Temp\Mathematica", "C:/GrSU_Books"}]**
Out[2556]= {False, True, True, True, True, True, True}

Considering the circumstance, that the ideology of the file organization of the computer quite allows in a number of cases of work with tools of access to identify datafiles and directories, this function is represented as an useful enough tool for both types of elements of file system of the computer. In a number of cases arises a necessity of reading out of a datafile entirely, excluding from its contents the symbols***"\r\n"*****–** carriage return and line feed. The following **ReadFullFile** procedure quite successfully solves this problem. The procedure call**ReadFullFile[*f*]** returns contents of a datafile***f*** with replacement of its symbols***"\r\n"*** onto symbols***"";*** if the datafile***f*** is absent in file system of the computer, the procedure call returns the$Failed. Whereas the call**ReadFullFile[*f, y*]** in addition through the second optional argument***y*** *–an undefinite variable–* returns a full name or a full path to the datafile***f;*** at that, if***y*** is a string, then***y*** replaces in the returned contents of the datafile***f*** all symbols***"\r\n"*** onto the string***y.*** The following fragment represents source code of the procedure along with examples of its usage.

In[2554] **:= ReadFullFile[f_ /; StringQ[f], y___] := Module[{a, b = $Art6Kr$}, If[FileExistsQ[f], a = f, ClearAll[$Art6Kr$]; If[! FileExistsQ1[f, $Art6Kr$], Return[$Failed], a = $Art6Kr$[[1]]]]; $Art6Kr$ = b; StringReplace[StringJoin[Map[FromCharacterCode, BinaryReadList[a]]], "\r\n"–> If[{y}!={}, If[StringQ[y], y, If[! HowAct[y], y =a; "", ""]], ""]]]**

In[2555] **:= ReadFullFile["Cinema.txt", t]**
Out[2555]= "http://100trav.com/ochishhenie-sosudov.html?utm_source= directadvert&utm_medium=ochishhenie-sosudov.html&utm_campaign= directadvert.ru http://www.worldlento4ka.com/russkiye-serialy/
http://www.worldlento4ka.com/7820-cherta-2014.html– 5
http://www.worldlento4ka.com/7830-ment-v-zakone-9-2014.html"
In[2556]**:= t**
Out[2556]= "C:\GrGU_Books\Cinema.txt"
In[2557]**:= ReadFullFile["AvZAgnVsvArtKr.doc"]**
Out[2557]= $Failed
In[2558]**:= ReadFullFile["DataFile.txt"]**
Out[2558]= "AvzAgnVsvArtKrRansIan2015"
In[2559]**:= ReadFullFile["DataFile.txt", " | "]**
Out[2559]= "Avz| Agn| Vsv| Art| Kr| Rans| Ian| 2015| " Once again it is necessary to remind that all elements of file system of the computer should be coded with the*separators* determined by the predefined ***$PathnameSeparator*** variable, by default as a separator the*double backslash* "\" is used. Meanwhile, in the**Mathematica** system in general the double backslash"\" and the slash"/" are distinguished as separators, namely**:** if the double backslash plays a part of standard separator of elements of file system of the computer, then the slash can also quite carry out this function, excepting a case when the slash is coded at the end of a chain of directories or at its use in a call of the**Run** function

as a whole. For elimination of the first situation we created the simple**DirQ** function considered above.

```
In[2567] := Map[DirectoryQ, {“C:\Program Files (x86)/Maple 11/”, “C:/Program
Files (x86)/Maple 11\”, “C:/Program Files (x86)/Maple 11”}]
Out[2567]= {False, True, True}
In[2568]:= Map[DirQ, {“C:\Program Files (x86)/Maple 11/”, “C:/Program Files
(x86)/Maple 11\”, “C:/Program Files (x86)/Maple 11”}]
Out[2568]= {True, True, True}
```

At that, the call **SetPathSeparator[*x*]** of a simple procedure makes setting of a separator**“\”** or**”/”** for paths to datafiles/directories for a period of the current session with returning of a new*separator* in string format as the next simple enough fragment rather visually illustrates.

```
In[2642]:= $PathnameSeparator
Out[2642]= “\”
In[2643]:= SetPathSeparator[x_ /; MemberQ[{“/”, “\”}, x]] := Module[{},
Unprotect[$PathnameSeparator]; $PathnameSeparator = x;
SetAttributes[$PathnameSeparator, Protected]]

In[2644] := {SetPathSeparator[“/”]; $PathnameSeparator,
SetPathSeparator[“\”]; $PathnameSeparator}
Out[2644]= {”/“, “\”}

In[2645] := StandPath[x_ /; StringQ[x]] := Module[{a, b = ””, c, k = 1},
If[MemberQ[Flatten[Outer[StringJoin, CharacterRange[“a”, “z”], {“:/“, “:\”}]], c =
ToLowerCase[x]], StringReplace[c, “/” –> “\”],

If[PathToFileQ[x], a = FileNameSplit[
StringReplace[ToLowerCase[ToLowerCase[x]], “/” –> “\”]];

For[k, k <= Length[a], k++, c = a[[k]]; If[! StringFreeQ[c, ” “], b = b <> StrStr[c] <>
“\”, b = b <> c <> “\”]]; StringTake[b, {1, –2}], ToLowerCase[x]]]]

In[2646] := StandPath[“C:/Program Files\Wolfram
Research/Mathematica/10.1/“]
Out[2646]= “c:\"program files"\"wolfram research"\ mathematica\10.1
In[2647]:= Map[StandPath, {“C:/“, “C:\”, “E:/”}]
Out[2647]= {”c:\”, “c:\”, “e:\”}
In[2648]:= StandPath[“AvzAgnVsvArtKt.TXT”]
Out[2648]= “avzagnvsvartkt.txt”
```

So, for the **Mathematica** system in most cases similar to the**Maple** system is also possible to use both types of separators of elements of a file system, however the told concerns only to**Windows** system, for other platforms the differences that in a number of cases are essential enough for programming are possible. As it was noted above, using different formats for names of the datafiles and full paths to them, we obtain an opportunity to open the same physical datafile in different streams, that in certain cases provides at times simplification of processing of datafiles. Meanwhile, in certain cases similar opportunity complicates the algorithms linked with processing of datafiles, for example, a datafile created on the basis of one format of name generally won't be recognized by standard means on the basis of another format: In[2652]:= **Write["RANS_IAN.txt"]; Close["Rans_Ian.txt"]**

General ::openx: Rans_Ian.txt is not open. >>
Out[2652]= Close["Rans_Ian.txt"]
In[2653]:= **Close["RANS_IAN.txt"]**
Out[2653]= "RANS_IAN.txt"

Thus, correct use of datafiles names and paths to them assumes, generally, work with the same format, as it illustrates the above example. Therefore as a quite simple reception allowing to unify names of datafiles/directories and paths to them it is possible to offer the following standard–*the symbols that compose names of datafiles and paths to them are coded in the lower case whereas as separators the double backslashes"\"are used.*

This problem is solved successfully by quite simple procedures **StandPath** and**FileDirStForm,** the source code of the first procedure with examples of application are represented in the previous fragment. So, the procedure call **StandPath[*x*]** in the above standardized format returns a datafile, directory or full paths to them. Moreover, the**StandPath** procedure for testing of an admissibility of an argument***x*** as a real path uses the**PathToFileQ** function presenting independent interest and providing the correctness of processing of the paths containing gap symbols. So, the usage by the**DirFD** procedure of the**StandPath** procedure allows to obtain quite correctly contents of any directory of file system of the computer which contains gap symbols and on which the***Dir*** command of***DOS*** system doesn't yield result as very visually the simple examples illustrate [30-33]. The**StandPath** procedure can be used rather effectively at development of different means of access in file system of the computer; moreover, the procedure is used by a number of means of access to the datafiles that are considered in the present book along with the means represented in the***AVZ_Package*** package [48].

The **Mathematica** system has two standard functions**RenameDirectory** and **RenameFile** for ensuring*renaming* of directories and datafiles of file system of the computer respectively. Meanwhile, from the point of view of the file concept these functions would be very expedient to be executed by uniform means because in this concept directories and datafiles are in many respects are identical and their processing can be carried out by the same means. At the same time the mentioned standard functions and on*restrictions* are quite identical, namely: for renaming of name***x*** of an element of file system onto a new name***y*** the element with the name***y*** has to be absent in the system, otherwise**$Failed** with a diagnostic message are returned. Moreover, if as***y*** only a new name without full path to a new element***y*** is coded, its copying into the current directory is made; in case of a directory***x*** it with all contents is copied into the current directory under a new name***y.*** Therefore, similar organization is rather inconvenient in many respects, what stimulated us to determine for renaming of directories and datafiles the uniform**RenDirFile** procedure which provides renaming of an element***x****(directory or datafile)* in situ with preservation of its type and all its attributes; at that, as argument***y*** a new name of the element***x*** is used. Therefore the successful procedure call **RenDirFile[*x, y*]** returns the full path to a renamed element***x.*** In the case of existence of an element***y*** the message"Directory/datafile<***y***> already exists" is returned. In other unsuccessful cases the procedure call returns the$Failed or is returned unevaluated. The next fragment represents source code of the **RenDirFile** procedure along with examples of its most typical usage.

```
In[2632]:= RenDirFile[x_ /; FileExistsQ[x] || DirectoryQ[x],
y_ /; StringQ[y]] := Module[{b = StandPath[StringTrim[x, {“/”, “\”}]], a =
If[FileExistsQ[x], RenameFile, RenameDirectory],

c = StandPath[StringTrim[y, {“/”, “\”}]]}, If[PathToFileQ[b] && PathToFileQ[c]
&& FileNameSplit[b][[1 ;;–2]] == FileNameSplit[c][[1 ;;–2]], Quiet[Check[a[b, c],
“Directory/datafile <” <> y <>”> already exists”]], If[PathToFileQ[b] && !
PathToFileQ[c],

Quiet[Check[a[b, FileNameJoin[Append[FileNameSplit[b][[1 ;;–2]], StringReplace[c,
{“/”–> ””, “\”–> ””}]]]], “Directory/datafile <” <> y <> “> already exists”]], If[!
PathToFileQ[b] && ! PathToFileQ[c], Quiet[Check[a[b, StringReplace[c, {“/”–> ””,
“\”–> ””}]], “Directory/datafile <” <> y <> “> already exists”]], $Failed]]]]

In[2633] := RenDirFile[“C:/Temp\Books.doc”, “Books_GrSU.doc”] Out[2633]=
“c:\temp\books_grsu.doc”
In[2634]:= RenDirFile[“C:/Temp/Noosphere Academy”, “Rans_Ian”] Out[2634]=
“c:\temp\rans_ran”
In[2635]:= RenDirFile[“C:\Temp/Kino Online.txt”, “Cinema Online.txt”] Out[2635]=
“c:\temp\cinema online.txt”
In[2636]:= RenDirFile[“RANS_IAN.txt”, “ArtKr.txt”]
Out[2636]= “C:\Users\Aladjev\Documents\artkr.txt”
In[2637]:= RenDirFile[“RANS_IAN.txt”, “ArtKr.txt”]
Out[2637]= RenDirFile[”RANS_IAN.txt“, “ArtKr.txt”]
In[2638]:= RenDirFile[“C:/Temp\agn”, “Agn”]
Out[2638]= “Directory/datafile<Agn> already exists”
In[2639]:= RenDirFile[“C:/Temp\Avz.doc”, “Agn.doc”]
Out[2639]= “c:\temp\agn.doc”
```

The special tools of processing of files and directories are considered below.

7.5. Certain special means of processing of datafiles and directories

In the given section some special tools of processing of directories and files are represented; in certain cases they can be useful enough. So, removal of a datafile in the current session is made by means of the standard**DeleteFile** function whose call**DeleteFile[{*x,y,z,…*}]** returns*Null*, i.e. nothing in case of successful removal of the given datafile or their list, and**$Failed** otherwise. At that, in the list of datafiles only those are deleted that have no***Protected*–** attribute. Moreover, this operation doesn’t save the deleted datafiles in the system***Recycle Bin*** directory, that in certain cases is extremely undesirable, first of all, in the light of possibility of their subsequent restoration. The fact that the system function**DeleteFile** is based on the***Dos*** command***Del*** that according to specifics of this operating system*immediately* deletes a datafile from file system of the computer without its preservation, that significantly differs from similar operation of the***Windows*** system that by default saves the deleted datafile in the special***Recycle Bin*** directory.

For elimination of similar shortcoming the **DeleteFile1** procedure has been offered, whose source code with examples of application are represented by the fragment below. The successful procedure call**DeleteFile1[*x*]** returns***0,*** deleting datafiles given by an argument***x*** with saving them in the***Recycle Bin*** directory of the***Windows*** system. Meanwhile, the datafiles removed by means of procedure call**DeleteFile1[*x*]** are saved in***Recycle Bin*** directory, however they are invisible to viewing by the system means, for example, by means of***Ms Explorer,*** complicating cleaning of the given system directory. Whereas the procedure call**DeleteFile1[*x, t*]** with the*2nd* optional argument ***t*** *–an undefinite variable–* thru it in addition returns the list of datafiles which for one reason or another were not removed. At that, in the system***Recycle Bin*** directory a copy only of the last deleted datafile always turns out. This procedure is oriented on***Windows XP*** and***Windows 7,*** however it can be spread to other operational platforms. The fragment below represents source code of the**DeleteFile1** procedure along with some examples of its usage.

For restoration from the system directory ***Recycler Bin*** of the packages that were removed by means of the**DeleteFile1** procedure on the***Windows XP*** platform, the**RestoreDelPackage** procedure providing restoration from the system directory***Recycler Bin*** of such packages has been offered [30,48]. The successful call**RestoreDelPackage[*F, "Context'"*],** where the first argument ***F*** determines the name of a file of the format {***"cdf", "m", "mx", "nb"***} that is subject to restoration whereas the second argument– the context associated with a package returns the list of full paths to the restored files, at the same time by deleting from the directory***Recycler Bin*** the restored datafiles with the necessary package. At that, this means is supported on the***Windows XP*** platform while on the***Windows 7*** platform the**RestoreDelFile** procedure is of a certain interest, restoring datafiles from the directory***Recycler Bin*** that earlier were removed by means of the**DeleteFile1** procedure.

The successful call **RestoreDelFile[*F,r*],** where the first argument***F*** defines the name of a datafile or their list that are subject to restoration whereas the second argument determines the name of a target directory or full path to it for the restored datafiles returns the list of paths to the restored datafiles**;** at the same time, the deleting of the restored files from the directory***Recycler Bin*** isn**'**t done. In the absence of the requested files in the directory***Recycler Bin*** the procedure call returns the empty list, i.e. {}. It should be noted that only nonempty datafiles are restored. If the second argument***r*** determines a directory name in string format, but not the full path to it, a target directory ***r*** is created in the active directory of the current session. The next fragment represents source code of the procedure along with examples of its usage.

On the other hand, for removal from the ***Recycle Bin*** directory of datafiles saved by means of the**DeleteFile1** procedure on the***Windows XP*** platform, the procedure is used whose call**ClearRecycler[]** returns***0,*** deleting files of the specified type from the system***Recycle Bin*** directory with saving in it of the datafiles removed by means of***Windows XP*** or its appendices. At last, the***Dick Cleanup*** command in***Windows XP*** in some cases completely does not clear the system***Recycler*** directory from files what successfully does the procedure call **ClearRecycler[*"ALL"*],** returning ***0*** and providing removal of all datafiles from the system***Recycler*** directory. In [30,48] it is possible to familiarize with source code of the**ClearRecycler** procedure and examples of its usage. On the***Windows 7*** platform the**ClearRecyclerBin** procedure provides removal from the system***Recycler*** directory of all directories and datafiles or only of those that are caused by the**DeleteFile1** procedure. The successful procedure call**ClearRecyclerBin[]** returns*Null***,** i.e. nothing, and provides removal from the system***Recycle Bin*** directory of directories and datafiles that are caused by the**DeleteFile1** procedure. While the procedure call**ClearRecyclerBin[*x*],** where***x*** *–a some expression*– also returns*Null***,** i.e. nothing, and provides removal from the system***Recycle Bin*** directory of all directories and datafiles whatever the cause of their appearance in the given directory. At that, the procedure call on the*empty****Recycler*** directory returns **$Failed.** The fragment below represents source code of the procedure along with typical examples of its usage.

In[2552] **:= DeleteFile1[x_ /; StringQ[x]||ListQ[x], y___] := Module[{d, p, t, a = Map[ToString, Flatten[{x}]], b, c = $ArtKr$}, b = If[! StringFreeQ[Ver[], ” XP “], FilesDistrDirs[BootDrive[][[1]] <> “:\Recycler”][[1]], p = 90; ClearAll[$ArtKr$]; If[FileExistsQ1[“$recycle.bin”, $ArtKr$], d = $ArtKr$[[1]], Return[$Failed]]; b = SortBy[Select[Flatten[FilesDistrDirs[d]], DirectoryQ[#] &], Length[#] &][[2]]]; $ArtKr$ = c; c = Map[StandPath, Map[If[StringFreeQ[#, “:”], Directory[] <> “\” <> #, #] &, a]]; t = Map[Run[“Copy /Y ” <> # <> ” ” <> If[p == 90, b <> FileNameSplit[#][[–1]], b[[1]]]] &, c]; t = Position[t, 1]; c = If[t != {}, MinusList[c, b = Extract[c, t]], Null]; If[t != {}&& {y}!= {}&& ! HowAct[y], Quiet[y = b], Quiet[y = {}]]; Map[{Attrib[#, {}], Quiet[DeleteFile[#]]}&, c]; 0]**

In[2553] **:= DeleteFile1[{“Buthday1.doc”, “c:/Mathematica\desktop1.ini”, “C:/Temp/Agn/cinema.txt”, “Help.txt”, “Cinema.txt”, “copy.txt”}, t67]**
Out[2553]= 0
In[2554]**:= t67**
Out[2554]= {**”**c**:**\temp\agn\cinema.txt**“,**
“c**:**\users\aladjev\documents\help.txt**“,**
“c**:**\users\aladjev\documents\cinema.txt**“, “**c**:**\users\aladjev\documents\copy.txt**”**}

In[2555] **:= DeleteFile1[{“AvzKr.m”, “AgnArt.nb”}]**
Out[2555]= 0
In[2556]**:= DeleteFile1[“C:/Documents and Settings/Cinema Online.txt”]** Out[2556]= 0

In[2642]**:= RestoreDelFile[f_ /; StringQ[f] || ListQ[f], r_ /; StringQ[r]] :=**

Module[{b = ToString[Unique[“ag”]], c, p = $ArtKr$, t = Map[StandPath, Flatten[{f}]], h}, ClearAll[$ArtKr$]; If[FileExistsQ1[“$recycle.bin”, $ArtKr$], d = $ArtKr$[[1]]; $ArtKr$= p, Return[$Failed]]; Run[“Dir ” <> d <> “/B/S/L > ” <> b]; If[EmptyFileQ[b], $Failed, Quiet[CreateDirectory[r]]; c = ReadList[b, String]]; DeleteFile[b]; h[x_, y_] := If[FileExistsQ[x] && SuffPref[x, “\” <> y, 2], CopyFileToDir[x, StandPath[r]], “Null”]; c = Select[Flatten[Outer[h, c, t]], ! SameQ[#, “Null”] &]]

In[2643] **:= RestoreDelFile[{“Books.txt”, “History.doc”}, “restore”]** Out[2643]= {}
In[2644]**:= RestoreDelFile[{“Cinema.txt”, “Buthday.doc”}, “restore”]** Out[2644]= {**”**restore\buthday.doc**“, “**restore\cinema.txt**”**}
In[2645]**:= RestoreDelFile[“Cinema.txt”, “c:/Temp/restore”]**
Out[2645]= {**”**c**:**\temp\restore\cinema.txt**”**}

In[2646]:= **RestoreDelFile[{“Cinema.txt”, “Buthday.doc”, “Grodno1.doc”,**

“Copy.txt”}, “C:/restore”]
Out[2646]= {”C:\restore\buthday.doc“, “C:\restore\cinema.txt“, “C:\restore\copy.txt”}
In[2660]:= **ClearRecyclerBin[x___] := Module[{a, c = $ArtKr$, d, p, b = ToString[Unique[“ag”]]}, ClearAll[$ArtKr$];**

If[! FileExistsQ1[“$recycle.bin”, $ArtKr$], $Failed, d = StandPath[$ArtKr$[[1]]]; $ArtKr$ = c; Run[“Dir ” <> d <> “/B/S/L > ” <> b]; p = ReadList[b, String]; DeleteFile[b]; If[p == {}, Return[$Failed], Map[If[{x}== {}, If[SuffPref[a = FileNameSplit[#][[–1]], “$”, 1] || a === “desktop.ini”, Null, Attrib[#, {}]; If[FileExistsQ[#], Quiet[Check[DeleteFile[#], DeleteDirectory[#, DeleteContents–> True]]], Quiet[Check[DeleteDirectory[#, DeleteContents–> True], DeleteFile[#]]]]], If[FileNameSplit[#][[–1]] == “desktop.ini”, Null, Attrib[#, {}];

If[DirQ[#], Run[“RD /S/Q ” <> #], Run[“Del /F/Q ” <> #]]]]&, p];]]]

In[2661] **:= ClearRecyclerBin[]**
In[2662]**:= ClearRecyclerBin[500]**
In[2663]**:= ClearRecyclerBin[]**
Out[2663]= $Failed

The given tools rather essentially expand the functions of the **Mathematica** software of restoration of datafiles of any type and directories, removed by means of***Windows,*** its applications, our procedure**DeleteFile1** along with effective enough cleansing of the system***Recycle Bin*** directory.

Meanwhile, a number of tools of processing of datafiles and directories was based on the**BootDrive** procedure which is correct for*Windows 2000|2003| NT|XP* while since*Windows****7,***it is necessary to use the**BootDrive1** function whose source code with an example is given below. The call**BootDrive1[]** returns the***3*–**element list, whose first element**–***homedrive,*** the second**–***the system catalog,*** the third element**–***type of the current operating system*.

In[4842] **:= BootDrive1[] := Mapp[Part, GetEnvironment[{“SystemDrive”, “SystemRoot”, “OS”}], 2]**
In[4843]**:= BootDrive1[]**
Out[4843]= {”C:**”, “**C:\Windows**“, “**Windows_NT”}
Furthermore, this function can be used for an operation system, supported by the**Mathematica.** At that the type of an operating system in some cases by the call**GetEnvironment[]** is returned incorrectly**;** the presented example concerns***Windows 7,*** but***Windows_NT*** has been received.

Values of the global variables **$System, $SystemID** and**$OperatingSystem** define the strings describing the current operational platform. Meanwhile, in a number of cases the specification of the current operational platform represented by them can be insufficient, in that case it is possible to use the **PCOS** procedure, whose call**PCOS[]** returns the***2*–**element list, whose first element determines the name of the computer owner, whereas the second element**–** the type of an operating platform. The fragment below represents source code of the**PCOS** procedure along with an example of its usage.

In[2593]:= {**$System, $SystemID, $OperatingSystem**}
Out[2593]= {"Microsoft Windows (64-bit)", "Windows-x86-64", "Windows"}
In[2594]:= **PCOS[] := Module[**{**a = ToString[Unique["agn"]], b**},

Run["SYSTEMINFO > " <> a]; b = Map[StringSplit[#] &, ReadList[a, String][[1 ;; 2]]]; DeleteFile[a]; b = Map[#[[3 ;;–1]] &, b]; {**b[[1]][[1]], StringReplace[ListToString[b[[2]], " "], "\"" –> ""]**}**]** In[2595]:= **PCOS[]**

Out[2595]= {"ALADJEV–PC", "Microsoft Windows 7 Professional"}

The next useful procedure bears the general character at operating with the devices of direct access and are useful enough in a number of applications, first of all, of the system character. The next procedure to a great extent is an analog of***Maple***–procedure***Vol_Free_Space*** which returns a volume of free memory on devices of direct access. The call**FreeSpaceVol[*x*]** depending on type of an actual argument***x*** which should define the logical name in string format of a device, returns simple or the nested list**;** elements of its sublists determine a device name, a volume of free memory on the volume of direct access, and the unit of its measurement respectively. In the case of absence or inactivity of the device***x*** the procedure call returns the message*"Device is not ready"***.** The next fragment represents source code of the**FreeSpaceVol** procedure along with typical examples of its usage.

In[2590]:= **FreeSpaceVol[x_ /; MemberQ3[Join[CharacterRange["a", "z"], CharacterRange["A", "Z"]], Flatten[**{**x**}**]]] :=**

Module[{**a = ToString[Unique["ag"]], b, c =**{}**, d = Flatten[**{**x**}**], k =1, t**}**, For[k, k <= Length[d], k++, t = d[[k]]; b = Run["Dir /S " <> t <> ":\" <> " > " <> a]; If[b != 0, AppendTo[c,** {**t, "Drive is not ready"**}**], b = ReadList[a, String][[–1]]; b = StringSplit[b][[–3 ;;–1]]; AppendTo[c,** {**t, ToExpression[StringJoin[Select[Characters[b[[1]]], IntegerQ[ToExpression[#]] &]]], b[[2]]**}**]]]; DeleteFile[a]; If[Length[c] == 1, c[[1]], c]]**

In[2591] **:= FreeSpaceVol["c"]**
Out[2591]= {"c", 442106667008, "bytes"}
In[2592]**:= FreeSpaceVol[**{**"c", "d", "e", "a"**}**]**
Out[2592]= {{"c", 442106667008, "bytes"}, {"d", 0, "bytes"},

{"e", 9890848768, "bytes"}, {"a", "Drive is not ready"}}

The following procedure facilitates the solution of the problem of use of the external**Mathematica** programs or operational platform. The procedure call **ExtProgExe[*x*, *y*, *h*]** provides search in file system of the computer of a {***exe*|** ***com***} file with the program with its subsequent execution on parameters***y*** of the command string. Both arguments***x*** and***y*** should be encoded in string format. Successful performance of the given procedure returns the full path to***"$TempFile$"*** datafile of*ASCII*–format containing result of execution of a program***x*,** and this datafile can be processed by means of standard means on the basis of its structure. At that, in case of absence of the datafile with the demanded program***x*** the procedure call returns**$Failed** while at using of the third optional argument***h*** *–an arbitrary expression–* the datafile with the program***x*** uploaded into the current directory determined by the call **Directory[],** is removed from this directory**;** also the datafile***"$TempFile$"*** is removed if it is*empty* or implementation of the program***x*** was terminated abnormally. The fragment below represents source code of the**ExtProgExe** procedure along with typical examples of its usage.

In[2558]:= **ExtProgExe[x_ /; StringQ[x], y_ /; StringQ[y], h___] := Module[{a = "$TempFile$", b = Directory[] <> "\" <> x, c},**

Empty::datafile = "Datafile $TempFile$ is empty; the datafile had been deleted."; If[FileExistsQ[b], c = Run[x, " ", y, " > ", a], c = LoadExtProg[x];

If[c === $Failed, Return[$Failed]]; c = Run[x, " ", y, " > ", a]; If[{h}!= {}, DeleteFile[b]]]; If[c != 0, DeleteFile[a]; $Failed, If[EmptyFileQ[a], DeleteFile[a]; Message[Empty::datafile], Directory[] <> "\" <> a]]]

>aladjev >aladjev >aladjev >aladjev >aladjev >aladjev >aladjev >aladjev >aladjev >aladjev >aladjev >aladjev console console console console console console console console console console console console

In[2559] **:= ExtProgExe["HostName.exe", "", 1]**
Out[2559]= "C:\Users\Aladjev\Documents\$TempFile$" In[2560]:= **ExtProgExe["Rans_Ian.exe", "", 1]**
Out[2560]= $Failed
In[2561]:= **ExtProgExe["tasklist.exe", " /svc ", 1]**
Out[2561]= "C:\Users\Aladjev\Documents\$TempFile$" In[2562]:= **ExtProgExe["systeminfo.exe", "", 1]**
Out[2562]= "C:\Users\Aladjev\Documents\$TempFile$" In[2563]:= **Select[Map[StringTake[#, {3,–1}] &, ReadList[Directory[] <>**

"\" <> "$TempFile$", String]], # != "" &]

Out[2563] = {"ALADJEV–PC", "Microsoft Windows 7 Professional", "Microsoft Corporation", "Multiprocessor Free", "Aladjev", "Microsoft", "00371-OEM-8992671-00524", "9.08.2014, 21:45:35", "6.03.2015, 14:03:18", "Dell Inc.", "OptiPlex 3020", "x64-based PC", "Dell Inc. A03, 14.04.2014", "C:\Windows", "C:\Windows\system32", "en-us; English (US)", "\Device\HarddiskVolume2", "et; Estonian", "(UTC+02:00) Helsinki, Kyiv, Riga, Sofia, Tallinn, Vilnius", "C:\pagefile.sys", "WORKGROUP", "\ALADJEV–PC"}

In[2564] **:= ExtProgExe["qprocess.exe", "*"]**
Out[2564]= "C:\Users\Aladjev\Documents\$TempFile$" In[2565]:= **k = 1; h = ""; While[! SameQ[h, EndOfFile],**

h = Read["$TempFile$", String];

Print[h]; k++]; Close["$TempFile$"]; 1 1772 taskhost.exe
1 1864 dwm.exe
1 1900 explorer.exe
1 2252 rtkngui64.exe
1 2272 ravbg64.exe
1 2308 igfxtray.exe
1 2316 hkcmd.exe
1 2344 igfxpers.exe
1 2352 igfxsrvc.exe

1 2408 fahwindow.exe
1 2424 avg-secure-s…
1 2816 skype.exe
>aladjev >aladjev >aladjev >aladjev >aladjev >aladjev >aladjev >aladjev >aladjev
>aladjev >aladjev >aladjev >aladjev >aladjev >aladjev >aladjev >aladjev >aladjev
>aladjev >aladjev EndOfFile console console console console console console console
console console console console console console console console console console console
console console 1 2956 iusb3mon.exe 1 2972 avgui.exe
1 3020 ishelper.exe 1 3104 ctfmon.exe
1 3948 firefox.exe
1 2416 plugin-conta… 1 4704 flashplayerp… 1 4768 flashplayerp… 1 1832 vprot.exe
1 2644 mathematica.exe 1 2304 mathkernel.exe 1 4436 mathkernel.exe 1 2032
totalcmd64.exe 1 4276 winword.exe 1 4208 splwow64.exe 1 3372 notepad.exe 1 3716
notepad.exe 1 4956 cmd.exe
1 4344 conhost.exe
1 4440 qprocess.exe

At last, the next procedure provides search in the given directory of chains of subdirectories and datafiles containing a string***x*** as own components. The call**DirFilePaths[*x*, *y*]** returns the***2***–element list whose first element is a list of full paths to subdirectories of a directory***y*** which contain components***x*** whereas the second element is the list of full paths to datafiles whose names coincide with a string***x*.**

In[3642] **:= DirFilePaths[x_ /; StringQ[x], y_: BootDrive1[][[1]]<>"*.*"] := Module[{c = {}, h, d = {}, b = ToString[Unique["avz"]], a = StringTrim[StandStrForm[x], "\"]}, Run["DIR /A/B/S " <> StandPath[y] <> " > " <> b]; h = ReadList[b, String]; DeleteFile[b]; Map[If[! StringFreeQ[StandPath[#], {"\" <> a <> "\", "\" <> a}], If[DirectoryQ[#], AppendTo[c, #], AppendTo[d, #]], Null]&, h]; {c, d}]** In[3643]**:= DirFilePaths["cinema.txt", "c:\Temp/"]**
Out[3643]= {{}, {"c:\temp\Cinema.txt"}}

In[3644] **:= DirFilePaths["CuteWriter.exe", "C:/Users/Aladjev/"]** Out[3644]= {{}, {"c:\users\aladjev\Downloads\CuteWriter.exe"}} In[3645]**:= DirFilePaths["CuteWriter.exe"]**
Out[3645]= {{}, {"C:\Users\Aladjev\Downloads\CuteWriter.exe"}}

In the absence of the second optional argument ***y*** the procedure call instead of it supposes**BootDrive1[][[1]] <> "*.*".** The previous fragment presents source code of the procedure with some examples of its use. In certain cases of access to file system the given procedure is an useful enough means.

In a number of problems of processing of file system of the computer along with work with datafiles the following**VolDir** procedure can present quite certain interest. The procedure call**VolDir[*x*]** returns the nested***2***–element list, whose*first* element determines the volume occupied by a directory***x*** in bytes whereas the*second* element determines the size of free space on a hard disk with the given directory. Whereas procedure call**DirsFiles[*x*]** returns the nested***2***–element list, whose first element defines the list of directories contained in a directory***x*,** including***x*,** and the second element defines the list of all datafiles contained in the given directory. The following fragment represents source codes of the above procedures with examples of their use.

```
In[3625]:= VolDir[x_ /; DirectoryQ[x] ||
MemberQ[Map[# <> “:” &, Adrive[]], ToUpperCase[x]]] :=

Module[ {a = ToString[Unique[“agn”]], b, c, d = StandPath[x]}, b = Run[“DIR /S ”
<> d <> ” > ” <> a]; If[b != 0, $Failed, c = Map[StringTrim, ReadList[a, String][[–2
;;–1]]]]; DeleteFile[a]; c = Map[StringTrim, Mapp[StringReplace, c, {“ÿ”–> ””,
“bytes”–> ”” , “free”–> ””}]]; ToExpression[Map[StringSplit[#][[–1]] &, c]]]
In[3628]:= Map[VolDir, {“c:/users/aladjev/downloads”, “e:/avz_package”}]
Out[3628]= {{2129356859, 442634944512}, {5944994, 9888374784}}

In[3655]:= DirsFiles[x_ /; DirectoryQ[x] ||
MemberQ[Map[# <> “:” &, Adrive[]], ToUpperCase[x]]] := Module[{a
=ToString[Unique[“ag”]], b ={x}, c ={}, d =StandPath[x], f},

If[Run[“DIR /A/B/S ” <> d <> ” > ” <> a] != 0, $Failed, f = ReadList[a, String];
DeleteFile[a]; Map[If[DirectoryQ[#], AppendTo[b, #], If[FileExistsQ[#], AppendTo[c,
#], Null]] &, f]; {b, c}]]

In[3656]:= DirsFiles[“C:\users\Aladjev\downloads”]
Out[3656]= {{”c:\users\Aladjev\downloads”}, …,
“c:\users\aladjev\downloads\Mathematica_10.1.0_WIN.zip”}}
```

By the by, it should be noted that at processing of the list structures of rather large size the unpredictable situations are quite possible [30-33].

So, the means presented in the given chapter sometimes rather significantly simplify programming of the tasks dealing with file system of the computer. Along with that these means extend functional means of access, illustrating a number of useful enough methods of programming of problems of similar type. These means in a number of cases very significantly supplement the standard access means supported by system, facilitating programming of a number of very important appendices dealing with the datafiles of various format. Our experience of programming of the access means that extend the similar means of the systems**Maple** and**Mathematica** allows to notice that *basic* access means of the**Mathematica** system in combination with its*global* variables allow to program more simply and effectively the user`s original access means. Moreover, the created access means possess sometimes by the significantly bigger performance in relation to the similar means developed in the environment of the **Maple** software. So, in the environment of the **Mathematica** system it is possible to solve the problems linked with rather complex algorithms of processing of datafiles while in the environment of the**Maple** system, first of all, in case of large enough datafiles the efficiency of such algorithms leaves much to be desired. In a number of appendices the means, presented in the present chapter along with other similar means from our package [48] are represented as rather useful, by allowing at times to essentially simplify programming. Meanwhile, it must be kept in mind, a whole series of the means that are based on the**Run** function and the***DOS*** commands generally can be nonportable onto other versions of the system and an operational platform, demanding the corresponding adaptation onto appropriate new conditions.

Chapter 8. The manipulations organization with the user packages in the*Mathematica*software

Similarly to the well -developed software the**Mathematica** is the extendable system, i.e. in addition to the built–in means that quite cover requirements of quite wide range of the users, the system allows to program those means that absent for the specific user in environment of the built-in language, and also to extend and correct standard means. Moreover, the user can find the missing means which are not built-in, in the numerous packages both in the packages delivered with the**Mathematica**, and separately existing packages for various applied fields. The question consists only in finding of a package necessary for a concrete case containing definitions of the functions**,**modules and other objects demanded for an application programmed in the system. A***package*** has the standard organization and contains definitions of various objects, somehow the functions, procedures, variables, etc., that solve well– defined problems. In return the**Mathematica** system provides a standard set of packages whose composition is defined by the concrete version of the system. For receiving of composition of the packages that are delivered with the current release of the**Mathematica** it is possible to use the procedure, whose the call**MathPackages[]** returns the list of names of packages, whose names with a certain confidence speak about their basic purpose. Whereas the call**MathPackages[*x*]** with optional argument***x*** *–an undefinite variable–* provides through it in addition return of the three–element list whose first element defines the current release of the**Mathematica** system, the second element– type of the license and the third element– a deadline of action of the license. The following fragment represents source code of the procedure along with examples of its most typical usage.

```
In[2590]:= MathPackages[h___] := Module[{c = $InstallationDirectory, b, a =
"$Kr18Art26$", d}, d = Run["Dir " <> StandPath[c] <> "/A/B/O/S >
$Kr18Art26$"];

If[d != 0, $Failed, d = ReadList[a, String]; DeleteFile[a]; b = Map[If[! DirectoryQ[#]
&& FileExtension[#] == "m", FileBaseName[#], "Null"] &, d]; b = Select[b, # !=
"Null" &]; b = MinusList[DeleteDuplicates[b], {"init", "PacletInfo"}]; If[{h}!=
{}&& ! HowAct[h], h ={$Version, $LicenseType,
StringJoin[StringSplit[StringReplace[DateString[ $LicenseExpirationDate], " "–> "*
"], "*"][[1 ;;–2]]]}]]; Sort[b]]

In[2591] := MathPackages[]
Out[2591]= {"AbelianGroup", "AbortProtect", "Abs", "AbsoluteDashing",
"AbsoluteOptions", "AbsolutePointSize", "AbsoluteThickness", "AbsoluteTime",
"accessodbc", "AccountData", …………………,
……………………………………………………………………….…….. "WriteDemo",
"WriteString", "Wronskian", "WSDL", "XBox", "XGRID", "XML", "XMLElement",
"XMLObject", "XMLSchema", "Xnor", "Xor", "ZapfDingbats", "ZernikeR",
"ZeroTest", "Zeta", "ZetaZero", "Zip", "ZipfDistribution", "ZTest", "ZTransform"}
In[2592]:= Length[%]
Out[2592]= 2563
In[2593]:= MathPackages[Sv]; Sv
```

Out[2593]= {”10.1.0 for Microsoft Windows (64-bit) (March 24, 2015)**“, “**Professional**“,** **“**Wed 21 Oct”}

From the given fragment follows that the **Mathematica** system of version ***10.1*** contains***2563*** packages oriented on various appendices, including the packages of strictly system purpose. Before use of means that are contained in a certain applied package, this package should be previously uploaded into the current session by means of the function call**Get[*Package*].**

8.1. Concept of the context, and its use in the software of the*Mathematica*system

The ***context*** concept has been entered into the program environment of the system for organization of operation with symbols which represent various objects*(modules****,****functions****,****packages****,****variables and so on)***,** in particular, in order to avoid the possible conflicts with the symbols of the same name. The main idea consists in that that the*full name* of an arbitrary symbol consists of two parts, namely**:** a context and a short name, i.e. the full name of some object has the next format**:*“context’short name”*** where the symbol<**’**> *(backquote)* carries out the role of some marker identifying a context in the software of the system. For example,***Avzagn’Vsv*** represents a symbol with the context ***Avzagn*** and with short name***Vsv.*** At that, with such symbols it is possible to execute various operations as with usual names**;** furthermore, the system considers***aaa’xyz*** and***bbb’xyz*** as various symbols. The most widespread use of context consists in its assignment to functionally identical or semantically connected symbols. For example,

AladjevProcedures`StandPath, AladjevProcedures`MathPackages

the procedures **StandPath** and**MathPackages** belong to the same group of the means associated with***“AladjevProcedures’”*** context that is ascribed to our***AVZ_Package*** package [48]. The current context is defined any moment of the system session, the context is in the global variable**$Context:** In[2562]**:= $Context**
Out[2562]= **“**Global`**”**

In the current **Mathematica** session the current context by default is defined as***“Global’”.*** While the global variable**$ContextPath** determines the list of contexts after the variable**$Context** for search of a symbol entered into the current session. It is possible to reffer to symbols from the current context simply by their short names**;** at that, if this symbol is crossed with a symbol from the list determined by the**$ContextPath** variable, the second symbol will be used instead of a symbol from the current context, for example**:**
In[2563]**:= $ContextPath**
Out[2563]= {**”**AladjevProcedures`**”, “**TemplatingLoader`**”, “**PacletManager`**”,**

“System`**”, “**Global`**”**}
Whereas the calls**Context[*x*]** and**Contexts[]** return the context ascribed to a symbol***x*** and the list of all contexts of the current session respectively**:**

In[2564] **:= Context[ActUcontexts]**
Out[2564]= **“**AladjevProcedures`**”**
In[2565]**:= Contexts[]**

Out[2565]= {"AladjevProcedures`", "AladjevProcedures`ActBFMuserQ`",
"AladjevProcedures`ActRemObj`", "AladjevProcedures`ActUcontexts`", …,
==
"WSMLink`", "XML`", "XML`MathML`", "XML`MathML`Symbols`",

" XML`NotebookML`", "XML`Parser`", "XML`RSS`", "XML`SVG`"} At that, by analogy with file system of the computer, contexts quite can be compared with directories. It is possible to determine the path to a datafile, specifying a directory containing it and a name of the datafile. At the same time, the current context can be quite associated with the current directory to datafiles of which can be referenced simply by their names. Furthermore, like file system the contexts can have hierarchical structure, in particular:

"Visualization`VectorFields`VectorFieldsDump`" **.** So, the path of search of a context of symbols in the**Mathematica** system is similar to a path of search of program files. At the beginning of the session the current context by default is***"Global'",*** and all symbols entered into the session will be associated with this context, except for the built–in symbols, for example,**Do,** which are associated with context***"System'".*** The path of search of contexts by default includes contexts for system–defined symbols. Whereas for the symbols removed by means of the **Remove** function, the context can't be defined, for example:
In[2565]:= **Avz := 500; Context["Avz"]**
Out[2565]= "Global`"
In[2566]:= **Remove["Avz"]; Context["Avz"]**

Context ::notfound: Symbol Avz not found. >>
Out[2566]= Context["Avz"]
At using of the contexts there is no guarantee that two symbols of the same name are available in various contexts. Therefore the**Mathematica** defines as a maximum priority the priority of choice of that symbol with this name, whose context is the first in the list which is defined by the global variable **$ContextPath.** Therefore, for the placement of such context in the beginning of the specified list it is possible to use the following simple construction: In[2568]:= **$ContextPath**
Out[2568]= {"AladjevProcedures`", "TemplatingLoader`", "PacletManager`",

" System`", "Global`"}
In[2569]:= **PrependTo[$ContextPath, "RansIanAvz`"]**
Out[2569]= {"RansIanAvz`", "AladjevProcedures`", "TemplatingLoader`",

" PacletManager`", "System`", "Global`"}
In[2570]:= **$ContextPath**
Out[2570]= {"RansIanAvz`", "AladjevProcedures`", "TemplatingLoader`",

" PacletManager`", "System`", "Global`"}
The next rather useful procedure provides assignment of the given context to a*definite* or*undefinite* symbol. The procedure call**ContextToSymbol1[*x, y, z*]** returns*Null,* i.e. nothing, providing assignment of a certain***y*** context to a symbol***x;*** at that, the third optional argument***z*** **–** the string, defining for***x*** the usage; at its absence for an*undefinite* symbol***x*** the usage– empty string, i.e."", while for a*definite* symbol***x*** the usage has view"Help on***x***". The next fragment presents source code of the**ContextToSymbol1**

procedure along with the most typical examples of its usage.

In[2725]:= **ContextToSymbol1[x_ /; AladjevProcedures`SymbolQ[x], y_ /; AladjevProcedures`ContextQ[y], z___] :=**

Module[{a, b = ToString[x]}, Off[General::shdw]; a = StringReplace["BeginPackage["AvzAgnVsvArtKr`"]\n 90::usage=73\nBegin["`90`"]\n500\nEnd[]\nEndPackage[]", {"AvzAgnVsvArtKr`" -> y, "73"–> If[AladjevProcedures`PureDefinition[x] === $Failed, """", If[{z}!= {}&& StringQ[z], AladjevProcedures`ToString1[z], AladjevProcedures`ToString1["Help on " <> b]]], "90"–> b, "500"–> If[AladjevProcedures`PureDefinition[x] === $Failed, b, AladjevProcedures`PureDefinition[x]]}]; Remove[x]; ToExpression[a]; On[General::shdw]]

In[2726] **:= Sv[x_] := Module[{a = 90, b = 500}, (a + b)*x^2]**
In[2727]**:= Context[Sv]**
Out[2727]= "Global`"
In[2728]**:= ContextToSymbol1[Sv, "Agn`"]**
In[2729]**:= Sv[73]**
Out[2729]= 3 144 110
In[2730]**:= ?Sv**

Help on Sv
In[2731]**:= Vsv[x_] := Module[{a = 500}, a*x]**
In[2732]**:= ContextToSymbol1[Vsv, "Tampere`", "Help on module Vsv."]** In[2733]**:= Context[Vsv]**
Out[2733]= "Tampere`"
In[2734]**:= ArtKr[x_] := Module[{a = 90, b = 500}, (a + b)*x]**

In[2734] **:= ContextToSymbol1[ArtKr, "AladjevProcedures`", "Help on module ArtKr."]**
In[2735]**:= DumpSave["C:/Users/Aladjev\Mathematica\Tampere.mx", "AladjevProcedures`"]**
Out[2735]= {"AladjevProcedures`"}
A new current session with the Mathematica system
In[3354]**:= Get["C:\Users\Aladjev\Mathematica\Tampere.mx"]**
In[3355]**:= ?? ArtKr**
Help on module ArtKr.
Art[x_]:=Module[{a=90,b=500},(a+b) x]
In[3356]**:= PureDefinition[Rans]**
Out[3356]= $Failed
In[3357]**:= ContextToSymbol1[Rans, "AgnVsv`"]**
In[3358]**:= Context[Rans]**
Out[3358]= "AgnVsv`"
In[3359]**:= $Packages**
Out[3359]= {"AgnVsv`", "HTTPClient`", "HTTPClient`OAuth`", …, "AladjevProcedures`", "Tampere`", "Agn`", …}
In[3360]**:= $ContextPath**

```
Out[3360]= {"AgnVsv`", "AladjevProcedures`", "Tampere`", "Agn`", ...}
```

At that along with possibility of assignment of the given context to symbols the **ContextToSymbol1** procedure is an useful enough means for extension by new means of the user package contained in a *mx* file. The technology of similar updating is as follows. On the *first* step a file ***x*** of ***mx*** format with the user's package having ***y*** context is uploaded into the current session by the function call **Get[*x*].** Then, in the same session the definition of a new means ***f*** with its usage ***u*** which describes the given means is evaluated. At last, by the procedure call **ContextToSymbol1[*f,y,u*]** the assignment of a ***y*** context to the symbol ***f*** along with its usage ***u*** is provided. Moreover, the usage ***u*** can be directly coded in the procedure call, or be determined by a certain string ***u.*** At last, by the function call **DumpSave[*x, y*]** the saving in the ***mx***–file ***x*** of all objects having ***y*** context is provided. Similar approach provides a rather effective mechanism of updating in the context of both *definitions* and *usages* of the means entering the user's package which is located in a ***mx***–file. Yet, the approach is limited by packages located in datafiles of ***mx***–format. As a result the symbols with the same *short name* whose contexts are located in the list defined by the **$ContextPath** variable further from the beginning, are inaccessible for access to them by means of their short names. Therefore for access to them it is necessary to use full names of the following format ***"Context'Name";*** furthermore, at entering into the current session of the new symbols *overlapping* the symbols of the same name of the list **$ContextPath** the corresponding message is output. Interesting enough questions in this context are considered enough in details in our books [30-33].

8.1.1. Interconnection of contexts and packages in the software of the *Mathematica* system

The *packages* are one of the main mechanisms of the **Mathematica** extension which contain definitions of the new symbols intended for use both outside of a package and in it. These symbols can correspond, in particular, to the new functions or objects determined in a package that extend the functional **Mathematica** possibilities. At that, according to the adopted agreement all new symbols entered in some package are placed in a context whose name is connected with the name of the package. At uploading of a package into the current session, the given context is added into the beginning of the list determined by the global variable **$ContextPath.** As a rule, for ensuring of association of a package with a context the construction **BeginPackage[*"x'"*]** coded at its beginning is used. At uploading of a package into the current session the context ***"x'"*** will update the current values of the global variables **$Context** and **$ContextPath.** Thus, our ***AVZ_Package*** package [48] contains **BeginPackage[*"AladjevProcedures'"*]** and at its uploading, the values of the specified variables accept the following view, namely**:**

```
In[2571]:= $ContextPath
Out[2571]= {"AladjevProcedures`", "TemplatingLoader`", "PacletManager`",

" System`", "Global`"}
In[2572]:= MemberQ[Contexts["*"], "AladjevProcedures`"]
Out[2572]= True
In[2573]:= $Packages
```

Out[2573]= {”HTTPClient`”,”HTTPClient`OAuth`”,

“ HTTPClient`CURLInfo`”, “HTTPClient`CURLLink`”, “JLink`”, “DocumentationSearch`”, “AladjevProcedures`”, “GetFEKernelInit`”, “TemplatingLoader`”, “ResourceLocator`”, “PacletManager`”, “System`”, “Global`”}

In[2574]:= **CNames[x_ /; ContextQ[x], y___] := Module[{b, a = Names[StringJoin[x, “*”]]}, b = Select[a, Quiet[ToString[Definition[ToString[#1]]]] != “Null” &];**

If[{y}!= {}&& PureDefinition[y] === $Failed, y = Sort[DeleteDuplicates[Select[a, PureDefinition[#] === $Failed &]]]]; Select[b, Attributes[#] != {Temporary}&& ToString[Definition[#]] != “Null” &]]

In[2575] **:= CNames[“AladjevProcedures`”]**
Out[2575]= {”AcNb“, “ActBFMuserQ“, “ActCsProcFunc“, “ActRemObj“, “ActUcontexts“, “AddMxFile“, “Adrive“, “Adrive1“, “Affiliate“, “Aobj“, “Aobj1“, “Args“, “Args1“, “ArgsBFM“, “ArgsTypes“, …
………………………………………………………………………….. “WhatType“, “WhatValue“, “WhichN“, “XOR1“, “$CallProc“, “$InBlockMod“, “$Line1“, “$Load$Files$”, “$ProcName“, “$ProcType“, “$TestArgsTypes“, “$TypeProc“, “$UserContexts”}
In[2576]:= **Length[%]**
Out[2576]= 683
In[2577]:= **CNames[“AladjevProcedures`”, h]; h**
Out[2577]= {”a“, “b“, “c“, “d“, “h“, “k“, “p“, “S“, “x“, “y“, “z”}
In[2578]:= **CNames[“System`”]**
Out[2578]= {”AASTriangle“, “AbelianGroup“, “AbortKernels“, “AbortProtect“, “Abs“, …, “$Urgent“, “$UserBaseDirectory“, “$UserName“, “$Version“, “$VersionNumber”}

At that, in the above fragment instead of return of the complete list defined by the call**Contexts[”*”]** to save space only testing of existence in it of the specified context is done. From the full list defined by the call**Contexts[“*”]** can be easily noticed that in it along with this context exist elements of a type ***“AladjevProcedures’Name’”*** that determine full names of all objects whose definitions are located in the***AVZ_Package*** package [48]. While the**CNames** procedure presented in the previous fragment allows to differentially obtain the lists of all short names in a package with the given context of both the definitions existing in it, and undefinite from the standpoint of the current session. So, the call**CNames[*x*]** returns the list of all short names in package with a context***x*,** that have definitions in it**;** whereas the call**CNames[*x*, *y*]** in addition through argument***y*** ***–****an undefinite variable***–** returns the list of all undefinite short names in the package with a context***x*.** Along with that, the analysis of the list, returned thru optional argument***y*** provides additional possibility of check of contents of the package relative to definiteness of all objects contained in it. The**CNames** procedure provides an easy way of the differentiated analysis of contents of the packages formalized in the form of the**Mathematica** documents of the formats {***“nb”***, ***“cdf”***}. The mechanism of contexts has a number of rather essential features that need to be taken into account during the work in the environment of the system, first of all, at use of the procedural paradigm. These features are considered rather in details in [33]. In particular, after uploading of a package into the current session all its objects will be associated with a context ascribed to the package while the objects of the same name, whose definitions are evaluated in the current **Mathematica** session are associated with the context***"Global'".*** For definition of*contexts* of symbols the**ContextDef** procedure can be used, whose call**ContextDef[*x*]** returns the list of the contexts associated with an arbitrary symbol***x.*** If symbol***x*** isn't associated with any context, the empty list is returned, i.e. {}. The following fragment represents source code of the **ContextDef** procedure along with typical examples of its usage.

In[3325] **:= Get["GSV.mx"]**
In[3326]**:= BeginPackage["RansIan`"]**
GSV::usage = "help on GSV."
Begin["`GSV`"]
GSV[x_, y_, z_] := Module[{a = 6}, x*y*z + a]
End[]
EndPackage[]
Out[3326]= "RansIan`"
Out[3327]= "help on GSV."
Out[3328]= "RansIan`GSV`"

In[3332]**:= GSV[x_Integer, z_Integer] := Module[{a = 90}, (x + z)*a]** In[3333]**:= ContextDef[x_ /; SymbolQ[x]] := Module[{a = $ContextPath, b = ToString[x], c, d, k, j = 1},**

While[j <= 2, c = {}; k = 1; Quiet[While[k <= Length[a], d = a[[k]] <> b; If[! SameQ[ToString[ToExpression["Definition[" <> d <> "]"]], "Null"], AppendTo[c, d]]; k++]]; j++]; c]

In[3334] **:= ContextDef[GSV]**
Out[3334]= {"RansIan`GSV", "avzransian500`GSV", "Global`GSV"} In[3335]**:= ProcQ[x_, y_] := x*y**
In[3336]**:= ContextDef[ProcQ]**
Out[3336]= {"RansIan`ProcQ", "AladjevProcedures`ProcQ"}
In[3337]**:= Definition[avzransian500`ProcQ]**
Out[3337]= ProcQ[x_, y_]:= x*y
In[3338]**:= Definition["Global`GSV"]**
Out[3338]= Global`GSV[x_Integer, z_Integer]:= Module[{a= 90}, (x+ z)*a] In[3339]**:= Definition["RansIan`GSV"]**
Out[3339]= RansIan`GSV[RansIan`GSV`x_, RansIan`GSV`y_,

RansIan `GSV`z_]:= Module[{RansIan`GSV`a= 6}, RansIan`GSV`x*RansIan`GSV`y*RansIan`GSV`z+ RansIan`GSV`a] In[3340]**:= $ContextPath**
Out[3340]= {"avzransian500`", "RansIan`", "AladjevProcedures`", "TemplatingLoader`", "PacletManager`", "System`", "Global`"}

Thus, at using of the objects of the same name, generally speaking, to avoid misunderstandings it is necessary to associate them with the contexts which have been ascribed to them.

8.2. Definition of the user packages, and their usage in the*Mathematica*software

The global variable**$Packages** defines the list of the contexts corresponding to all packages uploaded into the current session, for example**:**

In[2569] **:= $Packages**
Out[2569]= {**"**AladjevProcedures`**", "**GetFEKernelInit`**", …, "**Global`**"**} In[2570]**:= Get[ "C:\Avz_Package\Aladjev.m"]; $Packages** Out[2570]= {**"**Aladjev`**", "**AladjevProceduresAndFunctions`**", …, "**Global`**"**} As it was already noted, each uploading of a new package into the current session adds the context corresponding to it to the beginning of the list that is determined by the global**$Packages** variable. Generally speaking, in the presence of the loaded packages their means it is quite possible to consider as means at the level of the built–in means of the**Mathematica** system. In effect, quite essential number of functions of the**Mathematica** system was realized in the form of packages. Meanwhile, in the majority of versions of the system preliminary uploading of packages for receiving access to means, contained in them is required. The majority of the**Mathematica** versions is provided with a standard set of packages which contain definitions of very large number of functions. For their use, as a rule, the appropriate packages it is necessary to upload professedly into the current session.**Mathematica** has the mechanism of both preliminary loading, and automatic loading of packages as needed. Meanwhile here one very essential circumstance takes place, namely**:** the help on such package means aren**'**t reflected in the help **Mathematica** system, and it can be received, for example, by the call***?Name.*** Similar organization is completely inconvenient, in particular, significantly conceding to the mechanism of organization of the help**Maple** system [27]. The main forms of preservation of definitions of the objects are a document ***(notebook)*** and a package***(package)*** that are located in datafiles of formats {***cdf, nb***} and {***m, mx***} respectively. At the same time between them there is a certain distinction. If uploading of the first into the current session allows to work with it as the document*(look over****,****execute****,****edit****,****save)****,*** then the package is intended only for uploading into the current session. At that, documents partially or completely can be considered as the packages. In particular, for convenience of work with the***AVZ_Package*** package it is presented in*three* main platform–independent formats, namely {***cdf, nb, m***}. It should be noted that binary datailes of the***mx***format optimized for fast uploading into the current session are nonportable both between versions of the**Mathematica** system, and between operational platforms.

A package uploading into the current session. Generally, a typical package is provided with two types of symbols determining as the*exported* symbols, and symbols for*internal* usage. For distinction these symbols are associated with different contexts. The standard reception consists in definition of the exported symbols in a context with the name***Name'*** which corresponds to the package name. Then, at uploading of a package it supplements the list defined by the global**$ContextPath** variable for providing of the call of the symbols which are in this context by their*short names*. While the definitions of all symbols intended for internal use are located in a context with a name ***Package'Private'*** that isn**'**t added to the list**$ContextPath,** without allowing to get access to the symbols of such context by their short names. As a rule, for setting of contexts of a package and global variables**$ContextPath** and **$Context** the standard sequence of functions in the package is used:

BeginPackage[***"Package`"*]**–*the setting for a package of the current context* ***"Package'";***
F1::usage = ***"Help"*–***the help on the exported***F1****symbol;further allows to receive the help by means of calls***?F1****and***Information[*F1*];**
F2::usage = ***"Help"*–***the help on the exported***F2****symbol;further allows to receive the help by means of calls***?F2****and***Information[*F2*];**
==
Begin["*`Private`*"]–*the setting of the context****"'Private'"****for local symbols;*
F1[*args*] :=*Definition1;…*–*definitions of local and global symbols of package;*
F2[*args*] :=*Definition2;…*–*definitions of local and global symbols of package;*
==
End[]
EndPackage[]–*the closing bracket of the package;simultaneously adding the context****"Package'"****to the beginning of the list of***$ContextPath** *at package uploading into the current session.*

The previous fragment at the same time represents the typical scheme of a package. The package given below serves as an illustration of filling of this scheme, namely:

```
In[2565] := BeginPackage["Tallinn`"]
G::usage = "Function G[x, y]:= 73*x^2+ 67*y+ 47+ S[x, y]." Begin["`Private`"]
S[x_, y_] := x^3 + y^3
G[x_ /; IntegerQ[x], y_Integer] := 73*x^2 + 67*y + 47 + S[x, y] End[]
EndPackage[]

Out[2565] = "Tallinn`"
Out[2566]= "Function G[x, y]:= 73*x^2+ 67*y+ 47+ S[x, y]." Out[2567]=
"Tallinn`Private`"
Out[2570]= "Tallinn`Private`"
In[2572]:= {S[90, 500], G[90, 500]}
Out[2572]= {S[90, 500], 126353847}
In[2573]:= $ContextPath
Out[2573]= {"Tallinn`", "AladjevProcedures`", "TemplatingLoader`",

" PacletManager`", "System`", "Global`"}
In[2574]:= $Context
Out[2574]= "Tallinn`"
In[2575]:= Information[S]
Out[2575]= Tallinn`S
In[2576]:= Information[G]
Out[2576]= Function G[x, y]:= 73*x^2+ 67*y+ 47+ S[x, y].

G[Tallinn `Private`x_/; IntegerQ[Tallinn`Private`x], Tallinn`Private`y_Integer]:= 73
Tallinn`Private`x^2+ 67 Tallinn`Private`y+ 47+
Tallinn`Private`S[Tallinn`Private`x, Tallinn`Private`y]

In[2577] := Tallinn`Private`S[90, 500]
Out[2577]= 125 729000
```

In[2578]**:= $Packages**
Out[2578]= {**”**Tallinn`**”**, **“**AladjevProcedures`**”**, **“**HTTPClient`OAuth`**”**,

“ HTTPClient`CURLInfo`**”**, **“**HTTPClient`CURLLink`**”**, **“**HTTPClient`**”**,
“GetFEKernelInit`**”**, **“**TemplatingLoader`**”**, **“**ResourceLocator`**”**, **“**PacletManager`**”**,
“System`**”**, **“**Global`**”**}

We will note that the definition of help ***(usage)*** for the means exported by a package serves as a certain kind of indicator what exactly these means are exported by a package whereas definitions of means without usages define local symbols which outside of the package are invisible, however they can be used by both*local***,** and*global* symbols of the package. Such organization is simpler and in some cases is a little more preferable. So the organizational scheme of a package can be simplified, having assumed rather simple view, represented by the following fragment. The fragment visually illustrates the principle of formation of a package taking into account the made remarks.

BeginPackage[*“Package`”*]–***the setting for a package of the current context* ***“Package`”;*
F::usage =*“Help”* –*the help on the exported***F***symbols***;***further allows to receive the help by means of calls***?F***and***Information[F];**
Begin[*“`F`”*]–*the setting of a context***“‘*F*’”***for a global symbol;*
F[*Formal args*] =*Definition F;…–**definitions of global package symbols;*
V[*Formal args*] =*Definition V;…–**definitions of local package symbols;*
==
End[]
EndPackage[]–*the closing bracket of the package***;***simultaneously adding the context****“Package`”****to the beginning of the list of***$ContextPath** *at package uploading into the current session.*

Thus, programming of a package can be simplified by means of definition of local variables without usages corresponding to them while all exports of the package are defined by the usages corresponding to them as illustrates the following simple enough fragment, namely**:**

In[2590] **:= BeginPackage[“Tallinn73`”]**
G6::usage = “Function G73[x, y] := 72*x^2 + 67*y + 47 + S6[x, y].” Begin[“`G6`”]
S6[x_, y_] := x^4 + y^4
G6[x_ /; IntegerQ[x], y_Integer] := 72*x^2 + 67*y + 47 + S6[x, y] End[]
EndPackage[]

Out[2590] = **“**Tallinn`**”**
Out[2591]= **“**Function G6[x, y]**:**= 72*x^2+ 67*y+ 47+ S6[x, y]**.”** Out[2592]=
“Tallinn`G6`**”**
Out[2593]= **“**Tallinn`G6`**”**
In[2595]**:=** {**S6[90, 500], G6[90, 500]**}
Out[2595]= {S6[78, 460], 62566226747}
In[2596]**:= $ContextPath**
Out[2596]= {**”**Tallinn73`**”**, **“**Tallinn`**”**, **“**AladjevProcedures`**”**,

“ TemplatingLoader`**”**, **“**PacletManager`**”**, **“**System`**”**, **“**Global`**”**} In[2597]**:= $Packages**
Out[2597]= {**”**Tallinn73`**”**, **“**Tallinn`**”**, **“**AladjevProcedures`**”**,

"HTTPClient`OAuth`", "HTTPClient`CURLInfo`", "HTTPClient`CURLLink`", "HTTPClient`", "GetFEKernelInit`", "TemplatingLoader`", "ResourceLocator`", "PacletManager`", "System`", "Global`"}

In[2598] := **Information[S6]**
Global`S6
In[2598]:= **Information[G6]**
Function G6[x, y]:= 72*x^2+ 67*y+ 47+ S6[x, y].
G6[Tallinn73`G6`x_/; IntegerQ[Tallinn73`G6`x],
Tallinn73`G6`y_Integer]:= 72 Tallinn73`G6`x^2+
67 Tallinn73`G6`y+ 47+ Tallinn73`G6`S6[Tallinn73`G6`x, Tallinn73`G6`y]

So, the call **Context[*x*]** of the standard function returns a context associated with a symbol***x*.** Meanwhile, rather interesting question is determination of the***m***–file with a package containing the given context. The procedure call **FindFileContext[*x*]** returns the list of full paths to***m***–files with the packages containing the given context***x;*** in the absence of such datafiles the procedure call returns the empty list, i.e. {}. At that, the call**FindFileContext[*x, y, z, …*]** with optional arguments {***y,z, …***}– the names in string format of devices of external memory of direct access– provides search of required files on the specified devices instead of search in all file system of the computer in case of a procedure call with one actual argument. The search of the required***m***files is done also in the***Recycle Bin*** directory of the***Windows*** system as that very visually illustrates an example of the next fragment. It must be kept in mind that search within all file system of the computer can demand enough essential temporal expenditure.The next fragment represents source code of the**FindFileContext** procedure along with typical examples of its usage.

In[2600] := **FindFileContext[x_ /; ContextQ[x], y___] := Module[{b ={}, c = "", d = StringJoin["BeginPackage["", StrStr[x], "]"], s = {}, k = 1, j = 1, a = If[{y}== {}, Adrive[], {y}], f = "$Kr18_Art26$.txt"},**

While[k <= Length[a], Run["Dir ", StringJoin[a[[k]], ":*.*"], StringJoin[" /A/B/O/S > ", f]]; While[! c === EndOfFile, c = Read[f, String]; If[! DirQ[c] && FileExtension[c] == "m", AppendTo[b, c]]; j++]; c = ""; j = 1; k++]; k = 1; While[k <= Length[b], c = ToString[ReadFullFile[b[[k]]]]; If[! StringFreeQ[c, d], AppendTo[s, b[[k]]]]; k++]; DeleteFile[Close[f]]; s]

In[2601] := **FindFileContext["Tallinn`"]**
Out[2601]= {"C:\AVZ_Package\Tallinn.m"}
In[2602]:= **FindFileContext["AladjevProcedures`"]**
Out[2602]= {"C:\GrGU_Books\AVZ_Package\AVZ_Package.m",

" C:\Users\Aladjev\Mathematica\AVZ_Package.m"} In[2603]:= **FindFileContext["AvzAgnSvetArtKr`", "F"]**
Out[2603]= {}
In[2604]:= **FindFileContext["AladjevProcedures`"]**
Out[2604]= {"C:\$RECYCLE.BIN\S-1-5-21-2596736632-989557747-

1273926778 -1000\AVZ_Package.m",
"C:\GrGU_Books\AVZ_Package\AVZ_Package.m",

“C**:**\Users\Aladjev\Mathematica\AVZ_Package.m**”**}

For definition of the status of existence of a context *(absent context,a current context without file,a current context with a**m**–file,inactive context with a**m**–file)* the following**FindFileContext1** procedure can be used, whose source code with typical examples of usage represents the following fragment, namely**:**

In[2608]**:= FindFileContext1[x_ /; ContextQ[x]] :=**

Module[{**a = FindFileContext[x], b = If[MemberQ[$Packages, x], “Current”,** {}]}**, If[a != {}&& ! SameQ[b, {}], {b, a}, If[a != {}&& SameQ[b, {}], a, If[a == {}&& ! SameQ[b, {}], b, {}]]]]**

In[2609] **:= FindFileContext1[“Tallinn`”]**
Out[2609]= **“**Current**”**
In[2610]**:= FindFileContext1[“AladjevProcedures`”]**
Out[2610]= {**”**Current**“**, {**”**C**:**\$RECYCLE.BIN\S-1-5-21-2596736632-

989557747 -1273926778-1000\AVZ_Package.m**“**,
“C**:**\GrGU_Books\AVZ_Package\AVZ_Package.m**“**,
“C**:**\Users\Aladjev\Mathematica\AVZ_Package.m**”**}}

In[2611]**:= FindFileContext1[“Aladjev`”]**
Out[2611]= {**”**f**:**\avz_package\aladjev.m**”**}

In[2612] **:= FindFileContext1[“RansIanRacRea`”]**
Out[2612]= {}
In[2613]**:= FindFileContext1[“PacletManager`”]**
Out[2613]= {**”**Current**“**, {**”**C**:**\Program Files\Wolfram Research\

Mathematica\10.1\SystemFiles\Autoload\ PacletManager\PacletManager.m**”**}}
Depending on the status of a context***x*** the call**FindFileContext1[*x*]** returns the following result, namely**:**

– {**”**Current**“**, {***m***–files}}*–the current context**x**located in the indicated**m**–files;*
–”Current”*–the current context**x**,not associated with**m**–files;*
– {***m***–files}*–the context**x**is located in**m**–files,but not in the*$**Packages***list;*
– {}*–the context**x**is formally correct,but not actual.*

As an essential enough addition to the above procedures **FindFileContext** and**FindFileContext1** is the**ContextInFile** procedure providing search of datafiles of the types {***cdf,m, mx, nb, tr***} containing definitions of packages with the given context. The procedure call**ContextInFile[*x,y*]** returns the list of full paths to datafiles of the indicated types containing definitions of packages with a context***x*.** At that, search is executed in a directory, defined by the second optional argument***y;*** in its absence the search of datafiles is executed in the**“C:\”** directory. Return of the empty list, i.e. {}, determines absence of the sought-for datafiles in the given path of search. The fragment below represents source code of the procedure with examples of its usage.

In[2578]**:= ContextInFile[x_ /; ContextQ[x], y___] := Module[**{**b, d, h, Tav, c = “$Art26Kr18$”**}**, If[**{**y**}**!=** {}**&& DirectoryQ[y],**

Run[“DIR ” <> StandPath[y] <> “/A/B/O/S > $Art26Kr18$”], Run[“DIR C:\

/A/B/O/S > $Art26Kr18$”]]; d = ReadList[c, String]; DeleteFile[c];

Tav[t_ /; ListQ[t]] := Module[{m, v = {}, k, z,
a = “BeginPackage[” <> ToString1[x] <> “]”},

Map[If[FileExistsQ[#] && MemberQ[{“cdf”, “nb”, “m”, “mx”, “tr”}, FileExtension[#]], If[MemberQ[{“tr”, “m”}, FileExtension[#]] && ! StringFreeQ[ReadFullFile[#], a], AppendTo[v, #], If[MemberQ[{“cdf”, “nb”}, FileExtension[#]], {m, h, k}= {0, ””, 1}; For[k, k < Infinity, k++, h = Read[#, String]; If[h === EndOfFile, Close[#]; Break[], If[! StringFreeQ[h, “BeginPackage”] && ! StringFreeQ[h, x], m = 90; Close[#]; Break[], Continue[]]]]; If[m == 90, AppendTo[v, #], Null], If[FileExtension[#] == “mx”,

z = StringPosition[ReadFullFile[#], {“CONT”, “ENDCONT”}]; If[! StringFreeQ[StringTake[ReadFullFile[#], {z[[1]][[1]], z[[2]][[1]]}], ” ” <> x <> ” “]; AppendTo[v, #], Null]]]], Null] &, t]; v]; Tav[d]]

In[2579] **:= ContextInFile[“AladjevProcedures`”, “C:\Users\Aladjev\Mathematica”**]
Out[2579]= {”c:\users\aladjev\mathematica\AVZ_Package.cdf“, “c:\users\aladjev\mathematica\AVZ_Package.m“, “c:\users\aladjev\mathematica\AVZ_Package.mx“, “c:\users\aladjev\mathematica\AVZ_Package.nb”}
In[2580]**:= ContextInFile[“ArtKrSvetGal`”]**
Out[2580]= {}
In[2581]**:= ContextInFile[“AladjevProcedures`”, “E:\”**]
Out[2581]= {”e:\users\aladjev\mathematica\AVZ_Package.cdf“, “e:\users\aladjev\mathematica\AVZ_Package.m“, “e:\users\aladjev\mathematica\AVZ_Package.mx“, “e:\users\aladjev\mathematica\AVZ_Package.nb”}
In[2582]**:= ContextInFile[“PacletManager`”, “C:\Program Files\ Wolfram Research\Mathematica\10.1\SystemFiles\Autoload”]**
Out[2582]= {”c:\program files\wolfram research\mathematica\10.1 \systemfiles\autoload\PacletManager\PacletManager.m”}

The procedures **FindFileContext, FindFileContext1** and**ContextInFile** are rather useful during the operating with packages. Meanwhile, realization of search of files with the given context within all file system of the computer, as a rule, can demand enough essential time costs. Below, some other useful procedures for work with packages and their contexts will be represented. In a sense the procedures**ContextMfile** and**ContextNBfile** are inverse to the procedures **FindFileContext, FindFileContext1, ContextInFile,** their successful calls **ContextMfile[*x*]** and **ContextNBfile[*x*]** return the context associated with the package which is located in a datafile of formats***m*** and {***nb, cdf***} accordingly; the datafile is given by means of name or full path to it. The next fragment presents source codes of the procedures**ContextMfile** and**ContextNBfile** along with the most typical examples of their usage.

In[2570] **:= ContextMfile[x_ /; FileExistsQ[x] && FileExtension[x] == “m”] := Module[{b, a = ReadFullFile[x], c}, b = SubsString[a, {“BeginPackage["”, “"]”}]; c**

= If[b != {}, StringTake[b, {14,–2}]]; If[b === {}, $Failed, c = Flatten[StringSplit[c, “,”]]; c = Select[Quiet[ToExpression[c]], ContextQ[#] &]; If[Length[c] > 1, c, c[[1]]]]]]

In[2571] **:= ContextMfile[“c:\users\aladjev\mathematica\ AVZ_Package.m”]**
Out[2571]= “AladjevProcedures`”
In[2572]**:= ContextMfile[“D:\AVZ_Package\RansIan.m”]**
Out[2572]= $Failed
In[2573]**:= ContextMfile[“C:/AVZ_Package/AVZ_Package_1.m”]**
Out[2573]= “AladjevProcedures`”
In[2574]**:= ContextMfile[“C:/temp\A A A\Aladjev.m”]**
Out[2574]= “Aladjev`”
In[2575]**:= ContextMfile[$InstallationDirectory** <>
“\SystemFiles\Kernel\Packages\GraphEdit.m”]
Out[2575]= “GraphEdit`”

In[2580]**:= ContextNBfile[x_ /; FileExistsQ[x] && MemberQ[{“cdf”, “nb”}, FileExtension[x]]] := Module[{a = ””},**

While[! SameQ[a, EndOfFile], a = Read[x, String]; If[! StringFreeQ[a, “BeginPackage”], a = Quiet[ToExpression[ToExpression[StringSplit[a, “,”][[3]]]]]; Break[]]; Continue[]]; Close[x]; If[! ContextQ[a] || SameQ[a, EndOfFile], $Failed, a]]
In[2581]**:= ContextNBfile[“D:\AVZ_PACKAGE\AVZ_Package.nb”]** Out[2581]= “AladjevProcedures`”

In[2582] **:= ContextNBfile[“D:\AVZ_PACKAGE\Book_3.nb”]** Out[2582]= $Failed
In[2583]**:= ContextNBfile[“D:\AVZ_PACKAGE\AVZ_Package.cdf”]** Out[2583]= “AladjevProcedures`”
In[2584]**:= ContextNBfile[“C:/AVZ_Package/AVZ_Package_1.nb”]** Out[2584]= “AladjevProcedures`”
In[2585]**:= ContextNBfile[“C:/Temp/A A A\AVZ_Package.nb”]** Out[2585]= “AladjevProcedures`”

Thus, the **ContextNBfile** procedure similar to the**ContextMfile** procedure completes the previous fragment, but it is oriented onto the user’s packages located in datafiles of the format {***“cdf”, “nb”***} whose internal organization differs from the organization of***m***–files with packages. The procedure call **ContextNBfile[*x*]** returns the context associated with the package which is located in a datafile***x*** of the format {***“cdf”, “nb”***} that is given by means of a name or full path to it. If datafile***x*** doesn’t contain a context, the procedure call **ContextNBfile[*x*]** returns **$Failed.** Both procedures have a number of important enough appendices at work with datafiles containing packages.

On the basis of the **ContextMfile** procedure for testing of system packages *(**m**–files)* that are located in the directory defined by**$InstallationDirectory** variable the**SystemPackages** procedure has been created whose procedure call**SystemPackages[]** returns the list in which***2***–element sublists have the format {*Package,its context*} while the call**SystemPackages[*x*]** thru optional argument***x*** *–an undefinite variable–* in addition returns the list of the system packages which aren’t possessing contexts, i.e. are used for internal needs of the**Mathematica** system. The next fragment represents source code of the **SystemPackages** procedure along with typical examples of its usage.

```
In[2678]:= SystemPackages[y___] := Module[{a, b},
a = FileNames["*.m", $InstallationDirectory, Infinity]; b =
Quiet[DeleteDuplicates[Map[{FileBaseName[#],

ContextMfile[#] }&, a]]]; b = Select[b, # != {}&]; If[{y}!= {}&& ! HowAct[y], y =
Select[Map[If[SameQ[#[[2]], $Failed], #[[1]]] &, b], ! SameQ[#, Null] &]]; Select[b, !
SameQ[#[[2]], $Failed] &]]

In[2679]:= SystemPackages[]
Out[2679]= {{"Common", {"AuthorTools`Common`",
"AuthorTools`MakeProject`"}},
{"DiffReport", {"AuthorTools`DiffReport`",

" AuthorTools`Common`"}},
{"Experimental", "AuthorTools`Experimental`"},
{"ExportNotebook", {"AuthorTools`ExportNotebook`",

" AuthorTools`Common`"}},…,
============================================ {"WebpTools",
"WebpTools`"}, {"WebServices", "WebServices`"},

{ "DateString", "XMLSchema`DateString`"},
{"XMLSchema", "XMLSchema`"}}
In[2680]:= Length[%]
Out[2680]= 282
In[2681]:= SystemPackages[Sv]; Sv
Out[2681]= {"AstronomyConvenienceFunctionsLoader",
"AstronomyConvenienceFunctions", "PacletInfo", "Default",
"init", "DataDropClientLoader", "DataDropClient", …,
============================================
"DLL", "InstallNET", "JLinkCommon", "MakeNETObject",
"MathKernel", "NETBlock", "NET", "TerraService",
"WebServicesNavigator", "Implementation", "WSDL"} In[2682]:= Length[%]
Out[2682]= 2318
In[2683]:= t = TimeUsed[]; SystemPackages[Kr]; Kr; TimeUsed[]–t Out[2683]=
18.315
In[2684]:= Length[FileNames["*.*", $InstallationDirectory, Infinity]] Out[2684]= 24
793

In[2768] := t = TimeUsed[]; a = FileNames["*.*", "C:\", Infinity]; TimeUsed[]–t
General::dirdep: Cannot get deeper in directory tree: C:\Documents…>>
General::cdir: Cannot set current directory to PerfLogs. >>
General::cdir: Cannot set current directory to cache. >>
General::dirdep: Cannot get deeper in directory tree:C:\ProgramData…>> General::stop:
Further output of General::dirdep will be suppressed during this calculation. >>
General::cdir: Cannot set current directory to Favorites. >>
General::stop: Further output of General::cdir will be suppressed during this calculation.
>>
Out[2768]= 2.371
```

In[2769]:= **t = TimeUsed[]; Run["DIR C:\ /A/B/O/S > $Art26Kr18$"]; TimeUsed[]–t**
Out[2769]= 0.015
In[2770]:= **Length[a]**
Out[2770]= 165 672
In[2771]:= **t = ""; For[k = 1, k < Infinity, k++, If[t === EndOfFile, Break[], t = Read["$Art26Kr18$", String]; Continue[]]]; k**
Out[2771]= 191 242

Inasmuch as, in particular, the directory containing the system**Mathematica *10*** contains***24793*** datafiles of different types, their testing demands certain time needs as illustrates an example of the previous fragment. At the same time it must be kept in mind that in a view of the told, the access to internal packages of the**Mathematica** system by means of the*mechanism* of contexts is impossible. Here quite appropriate to make one rather essential remark.

Meanwhile, the **ContextMfile** procedure provides search only of the first context in a***m***–file with a package whereas generally multiple contexts can be associated with a package. The next**ContextMfile1** procedure provides the solution of this question in case of multiple contexts. The procedure call **ContextMfile1[*x*]** returns the list of the contexts or single context associated with a datafile***x*** of formats {***"m",“tr"***}, in case of lack of contexts the empty list, i.e. {} is returned. Furthermore, the additional***tr***–format allows to carry out search of contexts in the system datafiles containing contexts. Moreover, in case**FileExistsQ[*x*] =*False*** the search of a datafile***x*** is done in file system of the computer as a whole. Whereas the**ActUcontexts** procedure provides obtaining of the list of*contexts* of the current session that are associated with the user packages.
The procedure call**ActUcontexts[]** for obtaining of the list uses an algorithm that is based on the analysis of system datafiles of formats {***"m", "tr"***}, while the call**ActUcontexts[*x*]** where optional argument***x*** is arbitrary expression, is based on the search of system datafiles of the view**"StringTake[Context, {*1,–2*}]<>{*"m", "tr"*}".** If the first algorithm is more universal, whereas the second significantly more high–speed. The**ReadFullFile1** function used by the**ContextMfile1** procedure, is an useful modification of the**ReadFullFile** procedure. Whereas the procedure call**SysContexts[]** returns the list of all system contexts, and the function call**SystemSymbols[]** returns all system symbols. The fragment below represents source codes of the above means along with examples of their typical usage.

In[2600]:= **ContextMfile1[x_ /; MemberQ[{"m", "tr"}, FileExtension[x]]] :=**

Module[{b = "BeginPackage[", c, d, a = ReadFullFile1[If[FileExistsQ[x], x, Flatten[{FindFile1[x]}][[1]]]]}, If[a === {}, {}, c = StringPosition[a, b]; If[c == {}, {}, d = SubStrToSymb[StringTake[a, {Flatten[c][[2]],–1}], 1, "]", 1]; d = StringReplace[StringTake[d, {2,–2}], {"{"–> "", "}"–> ""}]; d = Map[ToExpression, StrToList[d]]; If[Length[d] == 1, d[[1]], d]]]] In[2601]:= **ContextMfile1["DocumentationSearch.m"]**

Out[2601] = {"DocumentationSearch`**", "**ResourceLocator`"}
In[2602]:= **ContextMfile1["IanRans.m"]**
Out[2602]= {}
In[2603]:= **ContextMfile1["AVZ_Package.m"]**

```
Out[2603]= "AladjevProcedures`"

In[2620]:= SubStrToSymb[x_ /; StringQ[x], n_ /; IntegerQ[n], y_ /; StringQ[y] && y
!= "", p_ /; MemberQ[{0, 1}, p]] :=

Module[ {a, b = StringLength[x], c, d, k}, If[n <= 0 || n >= b || StringFreeQ[x, y],
$Failed, c = StringTake[x, {n}]; For[If[p == 0, k = n–1, k = n + 1], If[p == 0, k >= 1, k
<= b], If[p == 0, k—, k++], If[Set[d, StringTake[x, {k}]] != y, If[p == 0, c = d <> c, c =
c <> d], Break[]]]; If[k < 1 || k > b, $Failed, If[p == 0, c = y <> c, c = c <> y]]]]
In[2620]:= SubStrToSymb["85123456786", 7, "8", 0]
Out[2620]= "8512345"

In[2740] := SubStrToSymb["85123456786", 7, "2", 1]
Out[2740]= $Failed
In[2620]:= SubStrToSymb["85123456786", 1, "6", 1]
Out[2620]= "85123456"

In[2649]:= ActUcontexts[x___] := Module[{c, d = {}, k, j,
a = MinusList[$Packages, {"System`", "Global`"}], b = FileNames[{"*.m", "*.tr"},
$InstallationDirectory, Infinity]},

c = DeleteDuplicates[Map[StringTake[#, {1, Flatten[StringPosition[#, "`"]][[1]]}] &,
a]]; If[{x}=={}, For[k =1, k <= Length[c], k++, For[j = 1, j <= Length[b], j++,
If[FileBaseName[b[[j]]] <> "`" == c[[k]] || MemberQ[ContextMfile1[b[[j]]], c[[k]]],
AppendTo[d, c[[k]]]; Break[]]]]; MinusList[c, d], c = Map[StringTake[#, {1,–2}] &,
c]; For[k = 1, k <= Length[c], k++, For[j= 1, j <= Length[b], j++,
If[FileBaseName[b[[j]]] == c[[k]],

AppendTo[d, c[[k]]]; Break[]]]]; MinusList[c, d]]] In[2650]:= ActUcontexts[590]
Out[2650]= {"Tallinn`", "Grodno`", "AladjevProcedures`"}
In[2656]:= ReadFullFile1[x_ /; FileExistsQ[x]] :=
StringReplace[Quiet[Check[ReadString[x], ""]], "\r\n"–> ""] In[2657]:=
ReadFullFile1["C:\Temp\Cinema.txt"]
Out[2657]= http://www.worldlento4ka.com/russkiye-serialy/

In[2670] := SysContexts[]:= Module[{a = Contexts[], b = ActUcontexts[590]},
Select[a, ! SuffPref[#, b, 1] &]]
In[2671]:= SysContexts[]
Out[2671]= {"Algebra`", "Algebraics`Private`", "Algebra`Polynomial`",….., "XML`",
"XML`MathML`", "XML`MathML`Symbols`", "XML`NotebookML`", "XML`Parser`",
"XML`RSS`", "XML`SVG`"}

In[2672]:= Length[%]
Out[2672]= 753
In[2694]:= SystemSymbols[] := Module[{a = Names["*"],
b = Join[Map[FromCharacterCode, Range[63488; 63596]], CNames["Global`"]], c =
ActUcontexts[590]}, MinusList[a, Join[b, Flatten[Map[CNames[#] &, c]]]]]
In[2695]:= h = SystemSymbols[]; Length[h]
Out[2695]= 6016
```

It should be noted that the above **ContextMfile1** procedure for the purpose of increase of performance significantly uses the**SubStrToSymb** procedure which belongs to means of processing of string expressions. The procedure call**SubStrToSymb[*x, n, y, p*]** returns a substring of a string***x*** bounded on the left***(p= 1)*** by a position***n*** and the first occurrence of a symbol***y,*** and on the right***(p= 0)*** by a position***n*** and the first occurrence of a symbol***y,*** i.e, at ***p = 0*** and***p = 1*** the search of the symbol***y*** is done right to left and left to right accordingly. Moreover, in a case of absence at search of a required symbol***y*** the procedure call**SubStrToSymb[*x, n, y, p*]** returns**$Failed,** while in other especial cases the procedure call is returned unevaluated. At that, the given procedure along with the above–mentioned application has enough much of other interesting appendices at processing of various string expressions.

The **Mathematica** system posesses the**FileNames** function which allows to obtain the list of datafiles of the given type in the specified directories of file system of the computer. In particular, our**SystemPackages** procedure uses this function for obtaining of***m***–files with system packages. Meanwhile, in the means considered earlier for operating with datafiles and directories the constructions of type**"Run[DIR …..]"** were generally used and that is why. First, in the case of large number of the tested files the considerable volume of***RAM*** is required while on the basis of the specified construction the list of datafiles is output into a***HD*** datafile. Secondly**–** the specified construction demands smaller time expenses concerning the**FileNames** function**;** at last, the function call on the main system directory causes erroneous situations, not allowing to receive the complete list of the datafiles contained in it. The last examples of the previous fragment illustrate the given reasons. For receiving access to package tools it is necessary that package containing them was uploaded into the current session, and the list determined by the **$ContextPath** variable has to include the context corresponding to the given package. A package can be loaded in any place of the current document by the function call**Get[*"context'"*]** or by the function call**Needs[*"context'"*]** to determine uploading of a package if the*context* associated with the package is absent in the list defined by the**$Packages** variable. In the case if package begins with**BeginPackage[*"Package'"*],** at its loading into the lists defined by the variables**$ContextPath** and**$Packages** only the context***"Package'"*** is placed, providing access to exports of the package and system tools. If the package uses means of other packages, the given package should begin with **BeginPackage[*"Package'",* {*"Package1'", …, "Package2'"*}]** with indication of the list of the contexts associated with such packages. It allows to include in addition in lists of**$ContextPath** and**$Packages** the demanded contexts. With features of uploading of packages the reader can familiarize in [33].

A package similarly to the procedures allows a nesting**;** at that, in the system all subpackages composing it are distinguished and registered. Moreover, the objects determined both in the main package, and in its subpackages are fully accessible in the current session after uploading of the nested package as quite visually illustrates the following very simple fragment. Meanwhile, for performance of the aforesaid it is necessary to redefine the**$ContextPath** variable after uploading of the nested package, having added all contexts of subpackages of the main package to the list determined by the variable**:**

```
In[2567] := BeginPackage["Kiev`"]
W::usage = "Help on W."
```

```
Begin["`W`"]
W[x_Integer, y_Integer] := x^2 + y^2
End[]
BeginPackage["Kiev1`", {"Kiev`"}]
W1::usage = "Help on W1."
Begin["`W1`"]
W1[x_Integer, y_Integer] := x*y + W[x, y]
End[]
EndPackage[]
EndPackage[]
```

```
Out[2567] = "Kiev`"
Out[2568]= "Help on W."
Out[2569]= "Kiev`W`"
Out[2571]= "Kiev`W`"
Out[2572]= "Kiev1`"
Out[2573]= "Help on W1."
Out[2574]= "Kiev1`W1`"
Out[2576]= "Kiev1`W1`"
In[2578]:= $ContextPath
Out[2578]= {"Kiev1`", "Kiev`", "System`"}
In[2579]:= $Packages
Out[2579]= {"Kiev1`", "Kiev`", "AladjevProcedures`", "GetFEKernelInit`",
" TemplatingLoader`", "ResourceLocator`", "PacletManager`", "System`", "Global`"}
In[2580]:= CNames["Kiev`"]
Out[2580]= {"W"}
In[2581]:= CNames["Kiev1`"]
Out[2581]= {"W1"}
In[2582]:= {W[42, 73], W1[42, 73]}
Out[2582]= {7093, 10159}
In[2583]:= Definition[W]
Out[2583]= W[Kiev`W`x_Integer, Kiev`W`y_Integer]:= Kiev`W`x^2+ Kiev`W`y^2
In[2584]:= Definition[W1]
Out[2584]= W1[Kiev`W1`x_Integer, Kiev`W1`y_Integer]:=
Kiev`W1`x Kiev`W1`y+ W[Kiev`W1`x, Kiev`W1`y]
```

After evaluation of definition of the user package of any nesting level it can be saved in datafiles of the following three system formats, namely:

F.nb* –** a datafile with the standard document*(notebook)* of the**Mathematica** system; moreover, there is a possibility of converting of such datafiles into datafiles of9*** formats, including formats {***"cdf", "m"***};
F.m* –** a datafile with a package of source format of the**Mathematica** system; ***F.mx* –** a datafile with a package inDumpSave*** format of the**Mathematica** system; this datafile is optimized under the used operational platform*(as a rule, **Windows, MacOSX, Linux**).*

As it was already noted above, the objects defined in the main package and in its subpackages are fully accessible in the current session after uploading of the main package into it, and also redefinition of**$ContextPath** variable by means of addition into the list determined by it, of all contexts associated with subpackages of the main package. In this context the**ToContextPath** procedure automates the given task, whose call**ToContextPath[*x*]** provides *updating* of contents of the current list determined by**$ContextPath** variable by means of adding to its end of all contexts of a***m***–file***x*** containing simple or nested package. So, the following fragment represents source code of the **ToContextPath** procedure along with a typical example of its usage.

In[5127] **:= ToContextPath[x_ /; FileExistsQ[x] && FileExtension[x] == "m"] := Module[{c, a = ReadFullFile[x], b = "BeginPackage["}, c = StrSymbParity[a, b, "[", "]"]; Map[If[! StringFreeQ[#, {"`"]", "`"}]"}], StringTake[#, {14,–2}]] &, c]; c = ToExpression[Flatten[Map[StringSplit[#, ","] &, c]]]; c = DeleteDuplicates[Map[If[ListQ[#], #[[1]], #] &, c]]; $ContextPath = DeleteDuplicates[Join[$ContextPath, c]]; $ContextPath]**

In[5128]**:= ToContextPath["C:\AVZ_Package\Kiev.m"]**
Out[5128]= {**"**AladjevProcedures`**", "**TemplatingLoader`**", "**PacletManager`**", "**System`**", "**Global`**", "**Kiev`**", "**Kiev1`**"**}

The successful procedure call **ToContextPath[*x*]** returns the updated value for**$ContextPath** variable. Taking into account the told, it is recommended to make uploading of a nested package***x(m–file)*** into the current session by means of the next pair of calls, namely**Get[*x*]; ToContextPath[*x*],** providing access to all means of the package***x.***

By the function call **Get[*"Name'"*]** the**Mathematica,**first of all, does attempt automatically to upload the version of the***"Name.mx"*** file that is optimized for the current platform if such file isn't found**,** the attempt is done to upload the***"Name.m"*** file which contains the code portable to other platforms. At that, it is supposed that a***m***–file with some package should be in one of the directories defined by the system**Path** variable. If a directory name is used, attempt to read the***"init.m"*** datafile intended for setting of packages of the directory is done. For providing the mode of automatic loading of packages the system**DeclarePackage** function is used. At the same time for removal of symbols of some context, more precisely, exports of a package with this context, the call**RemovePackage[*"Name'"*]** of our procedure is used.

As it was noted earlier, for each exported object of a certain package for it it is necessary to determine an***usage.*** As a result of uploading of such package into the current session all its*exports* will be available while the*local* objects, located in a section, in particular***Private,*** will be inaccessible in the current session. For testing of a package loaded into the current session or unloaded package which is located in a***m***–file regarding existence in it of global and local objects the following procedure**DefInPackage** can be used, whose call **DefInPackage[*x*],** where***x*** defines a datafile or full path to it, or the context associated with the package returns the nested list, whose the first element defines the package context, the second element– the list of local variables while the third element– the list of global variables of the package***x.*** If the argument***x*** doesn't define a package or a context, the call**DefInPackage[*x*]** is returned unevaluated. In case of an unusable context***x*** the procedure call returns**$Failed.** The fragment represents source code of the**DefInPackage** procedure along with the most typical examples of its usage.

```
In[2582] := BeginPackage["Kherson`"]
Gs::usage = "Help on Gs."
Ga::usage = "Help on Ga."
Vgs::usage = "Help on Vgs."
Begin["`Private`"]
W[x_, y_] := x + y
Vt[y_] := y + Sin[y]
Sv[x_] := x^2 + 23*x + 16
End[]
Begin["`Gs`"]
Gs[x_Integer, y_Integer] := x^2 + y^2
End[]
Begin["`Ga`"]
Ga[x_Integer, y_Integer] := x*y + Gs[x, y]
End[]
Begin["`Vgs`"]
Vgs[x_Integer, y_Integer] := x*y
End[]
EndPackage[];

Out[2582] = "Kherson`"
Out[2583]= "Help on Gs."
Out[2584]= "Help on Ga."
Out[2585]= "Help on Vgs."
Out[2586]= "Kherson`Private`"
Out[2590]= "Kherson`Private`"
Out[2591]= "Kherson`Gs`"
Out[2593]= "Kherson`Gs`"
Out[2594]= "Kherson`Ga`"
Out[2595]= "Kherson`Ga`"
Out[2596]= "Kherson`Vgs`"
Out[2598]= "Kherson`Vgs`"
In[2599]:= Map[FunctionQ, {Ga, Gs, Vgs, W, Vt, Sv}]
Out[2599]= {True, True, True, False, False, False}
In[2600]:= BeginPackage["Kherson1`"]

Gs1::usage = "Help on Gs1."
Ga1::usage = "Help on Ga1."
Begin["`Gs1`"]
Gs1[x_Integer, y_Integer] := x^2 + y^2
End[]
Begin["`Ga1`"]
Ga1[x_Integer, y_Integer] := x*y + Gs1[x, y]
End[]
EndPackage[];

In[2640]:= StringDependAllQ[s_String, a_ /; StringQ[a] || ListQ[a] && !
MemberQ[Map[StringQ, a], False]] := DeleteDuplicates[Map[StringFreeQ[s, #] &,
If[StringQ[a], {a}, a]]] == {False}

In[2641] := Map3[StringDependAllQ, "abcnq", {{"a","n","q"}, {"a","x","y"}}]
Out[2641]= {True, False}
In[2642]:= Map[! StringFreeQ["abcnq",#] &, {{"a","b","n","q"}, {"a","x","y"}}]
Out[2642]= {True, True}

In[2778]:= StringDependQ1[x_ /; StringQ[x], y_ /; ListStringQ[y]] :=

Module[ {a = x, b, k = 1}, For[k, k <= Length[y], k++, b = Flatten[StringPosition[a,
y[[k]]]]; If[b != {}, a = StringTake[a, {b[[2]] + 1,–1}], Return[False]]]; True]

In[2779] := Map3[StringDependQ1, "11abc222dcd3333xy44z6", {{"11", "222",
"333"}, {"11", "22222", "333"}, {"333", "44", "6"}}]
Out[2779]= {True, False, True}

In[2858] := MfilePackageQ[x_] := If[FileExistsQ[x] && FileExtension[x] == "m",
StringDependAllQ[ReadFullFile[x], {"(* ::Package:: *)", "(* ::Input:: *)", "::usage",
"BeginPackage["", "EndPackage[]"}], False]

In[2859] := MfilePackageQ["C:\AVZ_Package\AVZ_Package_1.m"] Out[2859]= True
In[2860]:= Map[MfilePackageQ, {"C:\AVZ_Package\66.nb", "Av.agn"}] Out[2860]=
{False, False}

In[2915]:= DefInPackage[x_ /; MfilePackageQ[x] || ContextQ[x]] := Module[{a, b =
{"Begin["`", "`"]"}, c = "BeginPackage["", d, p, g, t, k = 1, f, n = x}, Label[Avz];
If[ContextQ[n] && Contexts[n] != {}, f = "$Kr18Art26$";

Save[f, x]; g = FromCharacterCode[17]; t = n <> "Private`"; a = ReadFullFile[f, g];
DeleteFile[f]; d = CNames[n]; p = SubsString[a, {t, g}]; p =
DeleteDuplicates[Map[StringCases[#, t ~~ Shortest[___] ~~ "[" <> t ~~ Shortest[___]
~~ " := "] &, p]]; p = Map[StringTake[#, {StringLength[t] + 1,
Flatten[StringPosition[#, "["]][[1]]–1}]&, Flatten[p]]; {n, DeleteDuplicates[p], d},
If[FileExistsQ[n], a = ReadFullFile[n]; f = StringTake[SubsString[a, {c, "`"]"}],
{15,–3}][[1]]; If[MemberQ[$Packages, f], n =f; Goto[Avz]]; b =StringSplit[a, "*)(*"];
d = Select[b, ! StringFreeQ[StringReplace[#, " "–> ""], "::usage="] &]; d =
Map[StringTake[#, {1, Flatten[StringPosition[#, "::"]][[1]]–1}]&, d]; p =
DeleteDuplicates[Select[b, StringDependAllQ[#, {"Begin["`", "`"]"}] &]]; p =
MinusList[Map[StringTake[#, {9,–4}] &, p], {"Private"}]; t =
Flatten[StringSplit[SubsString[a, {"Begin["`Private`"]", "End[]"}], "*)(*"]]; If[t ==
{}, {f, MinusList[d, p], p},

g = Map[StringReplace[#, " " –> ""] &, t[[2 ;;–1]]]; g = Select[g, ! StringFreeQ[#,
":="] &]; g = Map[StringTake[#,

{1, Flatten[StringPosition[#, ":"]][[1]]–1}] &, g]; g =
Map[Quiet[Check[StringTake[#, {1, Flatten[StringPosition[#, "["]][[1]]–1}], #]] &,
g]; {f, g, d}], $Failed]]]

:= DefInPackage["Kherson1`"] In[2916]
Out[2916]= {"Kherson1`", {}, {"Ga1", "Gs1"}}}
In[2917]:= DefInPackage["C:\AVZ_Package\Kiev.m"]
```

Out[2917]= {"Kiev`", {}, {"W", "W1"}}
In[2918]:= **DefInPackage["C:\AVZ_Package\Kherson1.m"]** Out[2918]= {"Kherson1`", {"W1", "Vt1", "Sv1"}, {"Gs1", "Ga1", "Vgs1"}} In[2919]:= **DefInPackage["C:\AVZ_Package\Kherson.m"]** Out[2919]= {"Kherson`", {"Vt", "Sv", "W"}, {"Ga", "Gs", "Vgs"}} In[2920]:= **DefInPackage["Kherson`"]**
Out[2920]= {"Kherson`", {"Vt", "Sv", "W"}, {"Ga", "Gs", "Vgs"}}

For simplification of the **DefInPackage** procedure algorithm the expediency of additional definition of***2*** simple enough functions came to light, namely. The**StringDependAllQ** function expands the construction**!StringFreeQ** if is required a testing of belonging to a string of all substrings from the given list. The call**StringDependAllQ[*s*, *x*]** returns*True* only in the case if a string ***x*** is substring of a string***s***, or each string from the list***x*** belongs to a string***s.***

Whereas the procedure call **StringDependQ1[*x*, *y*]** returns*True* if a string***x*** contains an occurrence of a chain of the substrings determined by a list***y*** of strings and in the order defined by their order in the list***y***,otherwise*False* is returned. The given procedure has a number of important applications. At last, the function call**MfilePackageQ[*x*]** returns*True* only in the case if the string***x*** defines a real datafile of***m***format that is the standard package. The previous fragment represents source codes of both functions along with examples of their usage. It is supposed that local symbols of a package are in its section***Private,*** that is quite settled agreement. Meanwhile, qua of the local objects of a package act as well those for which usages aren't defined. So, the**DefInPackage** procedure successfully processes the packages with other names of local sections or without such sections at all, i.e. definitions of local symbols are located in a package arbitrarily. We leave the analysis of algorithm of the procedure as an useful exercise for the interested reader.

In a number of cases there is a need of full removal from the current session of the package uploaded into it. Partially the given problem is solved by the standard functions**Clear** and**Remove** however they don't clear the lists that are defined by variables**$Packages, $ContextPath** and by the call**Contexts[]** off the package information. This problem is solved by the**RemovePackage** procedure whose call**RemovePackage[*x*]** returns*Null,* i.e. nothing, at that, completely removing from the current session a package determined by a context***x,*** including all exports of the package***x*** and respectively updating the specified system lists. The following fragment represents source code of the**RemovePackage** procedure with the most typical examples of its usage.

In[2820]:= **RemovePackage[x_ /; ContextQ[x]] := Module[{a = CNames[x], b = ClearAttributes[{$Packages, Contexts}, Protected]},**

Quiet[Map[Remove, a]]; $Packages = Select[$Packages, StringFreeQ[#, x] &]; Contexts[] = Select[Contexts[], StringFreeQ[#, x] &];

SetAttributes[{$Packages, Contexts}, Protected]; $ContextPath = Select[$ContextPath, StringFreeQ[#, x] &];] In[2821]:= **$ContextPath**
Out[2821]= {"Kherson1`", "Kherson`", "AladjevProcedures`",

" TemplatingLoader`","PacletManager`", "System`", "Global`"} In[2822]:= **$Packages**
Out[2822]= {"Kherson1`", "Kherson`", "AladjevProcedures`",

“GetFEKernelInit`”, “TemplatingLoader`”, “ResourceLocator`”, “PacletManager`”, “System`”, “Global`”}

In[2823] := **Contexts[]**
Out[2823]= {”AladjevProcedures`”, “AladjevProcedures`ActBFMuserQ`”, “AladjevProcedures`ActRemObj`”, “AladjevProcedures`ActUcontexts`”, “AladjevProcedures`AddMxFile`”, “AladjevProcedures`Adrive1`”, **……**}
In[2824]:= **RemovePackage[“Kherson1`”]**
In[2825]:= **$Packages**
Out[2825]= {”Kherson`”, “AladjevProcedures`”, “GetFEKernelInit`”, “TemplatingLoader`”, “ResourceLocator`”, “PacletManager`”, “System`”, “Global`”}
In[2826]:= **Map[PureDefinition, {“Ga1”, “Gs1”}]**
Out[2826]= {$Failed, $Failed}

Meanwhile, it should be noted that the packages uploaded into the current session can have the objects of the same name; about that the corresponding messages are output. Qua of an active object acts the object whose context is in the list**$Packages** earlier, that quite visually illustrates the next fragment with the**RemovePackage** procedure usage. In this regard the procedure call **RemovePackage[*x*]** deletes a package with the given context***x*.**

In[2587] := **BeginPackage[“Pac1`”]**
W::usage = “Help on W.”
Begin[“`W`”]
W[x_Integer, y_Integer] := x^2 + y^2
End[]
EndPackage[]

Out[2587] = “Pac1`”
Out[2588]= “Help on W.”
Out[2589]= “Pac1`W`”
Out[2591]= “Pac1`W`”
In[2593]:= **BeginPackage[“Pac2`”]**

W::usage = “Help on W.”
Begin[“`W`”]
W[x_Integer, y_Integer] := x^3 + y^3
End[]
EndPackage[]

Out[2593] = “Pac2`”
W::shdw: Symbol W appears in multiple contexts {Pac2`,Pac1`}; definitions..
Out[2594]= “Help on W.”
Out[2595]= “Pac2`W`”
Out[2597]= “Pac2`W`”
In[2599]:= **$Packages**
Out[2599]= {”Pac2`”, “Pac1`”, “HTTPClient`”, “HTTPClient`OAuth`”,

“ HTTPClient`CURLInfo`”, “HTTPClient`CURLLink`”, “JLink`”, “DocumentationSearch`”, “AladjevProcedures`”, “GetFEKernelInit`”, “TemplatingLoader`”, “ResourceLocator`”, “PacletManager`”, "System`", "Global`"}

In[2600] := **W[90, 500]**
Out[2600]= 125 729000
In[2601]:= **Definition[W]**
Out[2601]= W[Pac2`W`x_Integer, Pac2`W`y_Integer]:= Pac2`W`x^3+

Pac2 `W`y^3 In[2602]:= **RemovePackage["Pac1`"]**
In[2603]:= **W[90, 500]**
Out[2603]= 125 729000
In[2604]:= **Definition[W]**
Out[2604]= W[Pac2`W`x_Integer, Pac2`W`y_Integer]:= Pac2`W`x^3+

Pac2 `W`y^3 In[2605]:= **RemovePackage["Pac2`"]**
In[2606]:= **Definition[W]**
Out[2606]= Null
In[2607]:= **$Packages**
Out[2607]= {"HTTPClient`", "HTTPClient`OAuth`",

" HTTPClient`CURLInfo`", "HTTPClient`CURLLink`", "JLink`",
"DocumentationSearch`", "AladjevProcedures`",
"GetFEKernelInit`", "TemplatingLoader`", "ResourceLocator`", "PacletManager`",
"System`", "Global`"}

A convenient enough way of packages saving is represented by the system **DumpSave** function, whose call**DumpSave[*F*, *x*]** returns the context***x*** of a package saved in a binary datafile***F*** in format optimized for its subsequent uploading into the**Mathematica** system. A package saved in the described way is loaded into the current session by means of the function call**Get[*F*]** with automatic activation of*all* definitions contained in it**;** at that, only*those* datafiles are correctly uploaded which were saved on the same computing platform by the**DumpSave** function of the**Mathematica** system.

Concerning the datafiles of ***mx***format with the user packages an interesting and useful problem of definition of the*context* and*objects***,** whose definitions are in the datafile of the given type, without its uploading into the current session arises. The**ContMxFile** procedure, whose source code with typical examples of use are presented by the fragment below, solves this problem.

In[2664] := **DumpSave["AVZ_Package.mx", "AladjevProcedures`"]** Out[2664]= {"AladjevProcedures`"}
In[2665]:= **DumpSave["Kherson1.mx", "Kherson1`"]**
Out[2665]= {"Kherson1`"}
In[2666]:= **DumpSave["Kiev.mx", "Kiev`"]**
Out[2666]= {"Kiev`"}

In[2667]:= **ContMxFile[x_ /; FileExistsQ[x] && FileExtension[x] == "mx", y___] := Module[{a = ReadFullFile[x], b = "CONT", c = "ENDCONT", d = "`", h, t},**

h = Flatten[StringPosition[a, {b, c}]][[1 ;; 4]]; h = StringReplace[StringTake[a, {h[[2]] + 1, h[[3]]–2}], ".10"–> ""]; h = StringJoin[Select[Characters[h], SymbolQ[#]&]] <> d; If[h == "", {}, If[MemberQ[$Packages, h] && {y}!= {}, {h,

CNames[h]}, If[! MemberQ[$Packages, h] && {y}!= {}, Quiet[Get[x]]; {{h, CNames[h]}, RemovePackage[h]}[[1]], t = SubsString[a, {h, "`"}]; t = Select[t, ! MemberQ[ToCharacterCode[#], 0] &]; {h, Sort[DeleteDuplicates[ Map[StringReplace[#, {h–> "", "`"–> ""}] &, t]]]}]]]]

In[2668] **:= ContMxFile["Kiev1.mx"]**
Out[2668]= {"Kiev1`", {"W", "W1", "W2", "W3"}}
In[2669]**:= ContMxFile["Kiev1.mx", 90]**
Out[2669]= {"Kiev1`", {"W", "W1"}}

In[2670] **:= ContMxFile["E:\AVZ_Package\AVZ_Package.mx"]** Out[2670]= {"AladjevProcedures`", {"ActBFMuserQ", "ActRemObj", "ActUcontexts", "AddMxFile", "Adrive1", "Affiliate", "Aobj", "Args", "ArgsBFM", "ArgsTypes", "Arity", "ArityBFM", ...}}
In[2671]**:= Length[%[[2]]]**
Out[2671]= 425
In[2672]**:= ContMxFile["Tallinn.mx"]**
Out[2672]= {"Grodno`", {"Gs", "Gs1", "Vgs", "Vgs1"}}

The procedure call**ContMxFile[*x*]** returns the nested list whose*first* element defines the context associated with the package contained in a***mx***–datafile***x*** while the*second* element determines the list of names in string format of all objects of this package irrespectively from existence for them of usages, i.e. of both local, and global objects. While the procedure call**ContMxFile[*x,y*],** where argument***y*** *–an arbitrary expression–* returns the nested list of similar structure, but with that difference that its second element defines the list of names of the objects of this package that are supplied with usages, i.e. only of the global objects. Withal, it should be noted that**ContMxFile** procedure presented in the previous fragment is intended for usage with the***mx***–files created on platform***Windows XP/7 Professional,*** its use for other platforms can demand the appropriate adaptation. The reason of it consists in that the algorithm of the**ContMxFile** procedure is based on an analysis of structure of***mx***–files that depends on platform used at creation of such datafiles. The following procedure**ContMxFile1** is an useful enough modification of the previous**ContMxFile** procedure which also uses an analysis of structure of ***mx***–files which depends on platform used at creation of such datafiles. The procedure call **ContMxFile1[*x*]** returns the nested list whose first element defines the context associated with the package contained in a***mx***–datafile***x*** while the*second* element determines the list of names in string format of all objects of this package irrespectively from existence for them of usages, i.e. local and global objects. Furthermore, similarly to the previous**ContMxFile** procedure the returned names determine objects whose definition returned by the call**Definition** contains the context. At that, is supposed that a file***x*** is recognized by the**FileExistsQ** function. The procedure algorithm enough essentially uses the function whose call**StrAllSymbNumQ[*x*]** returns*True* if a string***x*** contains only symbols and/or integers, and*False* otherwise. The fragment below represents source codes of both means along with examples of its typical usage.

In[2769] **:= ContMxFile1[x_ /;FileExistsQ[x] && FileExtension[x] =="mx"]:= Module[{a = ReadFullFile[x], b = "CONT", c = "ENDCONT", d, h, t}, h = Flatten[StringPosition[a, {b, c}]][[1 ;; 4]]; h = StringReplace[StringTake[a, {h[[2]] +**

1, h[[3]]–2}], ".10"–> ""]; h = StringJoin[Select[Characters[h], SymbolQ[#] &]] <> "`"; If[h == "", {}, d = StringPosition[a, h][[2 ;;–1]]; d = Map[StringTrim[#, "`"] &, Map[SubStrToSymb[a, #[[2]] + 1, "`", 1] &, d]]; {h, Sort[Select[d, StrAllSymbNumQ[#] &]]}]]

In[2770] **:= ContMxFile1["c:\users\mathematica\avz_package.mx"]** Out[2770]= {"AladjevProcedures`", {"ActBFMuserQ", "ActCsProcFunc", "ActRemObj", "ActUcontexts", "AddMxFile", "Adrive1", ..., "WhatType", "WhichN", "XOR1", "$ProcName", "$TypeProc"}}
In[2771]**:= Length[%[[2]]]**
Out[2771]= 427

In[2772]**:= StrAllSymbNumQ[x_ /; StringQ[x]] :=**
! MemberQ[Map[SymbolQ[#] ||Quiet[IntegerQ[ToExpression[#]]] &, Characters[x]], False] In[2773]**:= Map[StrAllSymbNumQ, {"PosListTest1", "BitGet`"}]** Out[2773]= {True, False}

The procedures **ContMxFile** and**ContMxFile1** adjoin the procedure, whose call**PackageMxCont[*x*]** returns the context of a***mx***–file**x;** the procedure call **PackageMxCont[*x, y*]** thru the*2nd* optional argument–*an undefinite variable* ***y*** **–** returns the nested list whose*first* element defines the list of*local* symbols whereas the*second* element defines the list of*global* symbols of the package that contained in the***mx***–file***x*** [33,48]. On***mx***–files without context or*local*/ *global* symbols the procedure call**PackageMxCont[*x*]** returns**$Failed** or the empty list accordingly, i.e. {}, for example**:** In[2728]**:=** {**PackageMxCont["E:\Avz_package\Avz_package.mx", s], s**} Out[2728]= {"AladjevProcedures`", {{}, {"AcNb", "ActBFMuserQ", "ActCsProcFunc", "ActRemObj", "ActUcontexts", "AddMxFile", "Adrive", "Adrive1", "Affiliate", "Aobj", "Aobj1", "Args", ...}}}

In[2729] **:= Length[%[[2]][[2]]]**
Out[2729]= 684
In[2730]**:=** {**PackageMxCont["PureDefinition.mx", s1], s1**}
Out[2730]= {$Failed**,** s1}

The procedure also is oriented on the platform ***Windows XP Professional*** in general, however on***Windows 7*** correctly returns the list of global symbols. In particular, for the platform***Windows 7 Professional*** the algorithm of the previous**ContMxFile** procedure is modified in the corresponding manner, taking into account the internal structure of the***mx***–datafiles created on the specified platform. This algorithm is realized by the procedure**ContMxW7,** whose call**ContMxW7[*x*]** returns the nested list whose first element defines the context connected with the package contained in a***mx***file***x*** whereas the second element defines the list of names in string format of*all* global objects of the package whose definitions contains a context ascribed to the package. Whereas on a***mx***–file without context the procedure call returns**$Failed.** At that, is supposed that a file***x*** is recognized by the**FileExistsQ** function. The fragment below represents source code of the**ContMxW7** procedure along with typical examples of its usage.

In[2633] **:= ContMxW7[x_ /; FileExistsQ[x] && FileExtension[x] == "mx"] :=**
Module[{a = FromCharacterCode[Select[BinaryReadList[x], # != 0 &]], b =

"CONT", c = "ENDCONT", d = "`", h, t, g = {}, k, f, n}, h = StringPosition[a, {b, c}] [[1;; 2]]; If[h[[1]]–h[[2]] == {–3, 0}, $Failed, t = StringTrim[StringTake[a, {h[[1]][[2]] + 2, h[[2]][[1]]–2}]]; a = StringTake[a, {h[[2]][[2]] + 1,–1}]; f = StringPosition[a, t]; Map[{c = "", For[k = #[[2]] + 1, k <= StringLength[a], k++, n = StringTake[a, {k, k}]; If[n == d, Break[], c = c <> n]]; If[StringFreeQ[c, StringTake[t, {1,–2}]], AppendTo[g, c], Null]}&, f]; {t, Select[Sort[g], StrAllSymbNumQ[#] &]}]]

In[2634] **:= ContMxW7["c:/users/aladjev/mathematica/AVZ_Package.mx"]**
Out[2634]= {"AladjevProcedures`", {"ActBFMuserQ", "ActRemObj", "ActUcontexts", "AddMxFile", "Adrive1", "Affiliate", "Aobj", "Aobj1", "Args", "Args1", "ArgsBFM", "ArgsTypes", "Arity", ..., "VizContentsNB", "VizContext", "WhatObj", "WhatType", "WhichN", "XOR1", "$ProcName", "$TypeProc"}}

In[2635] **:= Length[%[[2]]]**
Out[2635]= 427
In[2636]**:= ContMxW7["C:\users\mathematica\PureDefinition.mx"]** Out[2636]= $Failed

Unlike the above procedures **ContMxFile** and**ContMxFile1,** the following **ContMxFile2** procedure is based on another algorithm whose essence is as follows. First of all the existence in a***mx***file***x*** of a package is checked**;** at its absence$Failed is returned. Then upload in the current session of a package containing in the***mx***file***x*** is checked. At positive result the required result without unloading of a package***x*** is returned, otherwise the required result with unloading of a package is returned. In both cases a call**ContMxFile2[*x*]** returns the***2***–element list, whose first element determines a package context whereas the second**–** the list of names in string format of means, contained in the package. The procedure essentially uses the**IsPackageQ** procedure.

In[2878] **:= ContMxFile2[x_ /; FileExistsQ[x] && FileExtension[x] =="mx"]:= Module[{a = $Packages, b = "AvzAgnVsvArtKr`", c, h, g, d = Unique["ag"]}, h = ToString[d]; g = IsPackageQ[x, d];**

If[g === $Failed, $Failed,
If[g === True, {d, AladjevProcedures`CNames[d],

ToExpression["Remove[" <> h <> "]"] },
ToExpression["InputForm[BeginPackage["AvzAgnVsvArtKr`"]; EndPackage[]]"]; Off[General::shdw]; Get[x]; c = $Packages[[1]];

b = {c, AladjevProcedures`CNames[c]}; AladjevProcedures`RemovePackage[c]; On[General::shdw]; b]]]

In[2879] **:= ContMxFile2["c:\users\ mathematica\avz_package.mx"]** Out[2879]= {"AladjevProcedures`", {"AcNb", "ActBFM", "ActBFMuserQ",..., "$TestArgsTypes", "$TypeProc", "$UserContexts"}}

In[2880]**:= Length[%[[2]]]**
Out[2880]= 684
In[2918]**:= IsPackageQ[x_ /; FileExistsQ[x] && FileExtension[x] == "mx", y___] := Module[{a = ReadFullFile[x], b = "CONT", c = "ENDCONT", d, g = $Packages},**

If[! StringContainsQ[a, "CONT" ~~ __ ~~ "ENDCONT"], $Failed, d =

StringPosition[a, {b, c}][[1 ;; 2]]; d = StringTake[a, {d[[1]][[2]] + 1, d[[2]][[1]]–1}]; d =Select[Map[If[! StringFreeQ[d, #], #, Null] &, g], ! SameQ[#, Null]&]; If[{y}!= {}&& ! HowAct[y], y = If[d == {}, {}, d[[1]]], Null]; If[d != {}, True, False]]]

In[2919] := {**IsPackageQ["c:\users/mathematica/avz_package.mx", y6], y6}**
Out[2919]= {True**,** "AladjevProcedures`"}
In[2920]**:= IsPackageQ["PureDefinition.mx"]**
Out[2920]= $Failed
In[2921]**:= IsPackageQ["c:\users/aladjev\mathematica\Tallinn.mx"]** Out[2921]= False

The **IsPackageQ** procedure is intended for testing of any***mx***file regarding existence of the user**'**s package in it along with upload of such package into the current session. The call of the**IsPackageQ[*x*]** procedure returns$Failed if the***mx***–file doesn**'**t contain a package,*True* if the package which is in the ***mx***–file***x*** is loaded into the current session, and*False* otherwise. Moreover, the procedure call**IsPackageQ[*x, y*]** through the second optional argument ***y*** **–** an undefinite variable**–** returns the context associated with the package uploaded into the current session. In addition, is supposed that a datafile***x*** is recognized by the testing function**FileExistsQ,** otherwise the procedure call is returned unevaluated. The previous fragment represents source codes of both procedures**ContMxFile2** and**IsPackageQ** along with more typical examples of their usage.

Meanwhile, the **DumpSave** function has one rather essential shortcoming, namely**:** it saves contexts which are only formally contexts, i.e. correspond to them only by the format. In this connection the**DumpSaveP** function is more preferable, whose call**DumpSaveP[*f, x*]** provides saving in a datafile***f*** of the package with a context***x*** on condition that this package contains the *global* symbols**;** otherwise the**DumpSaveP** function call returns$Failed. The fragment represents source code of the function and examples of its usage.

In[3342]**:= PackageQ[x_ /; ContextQ[x]] := If[CNames[x] != {}, True, False]**
In[3343]**:= DumpSaveP[f_/; StringQ[f], x_ /; ContextQ[x]]:= If[PackageQ[x], DumpSave[f, x], $Failed]**

In[3344] **:= DumpSave["AVZ_Package.mx", "AladjevProcedures`"]** Out[3344]= {"AladjevProcedures`"}
In[3345]**:= RemovePackage["AladjevProcedures`"]**
In[3346]**:= Map[Definition, {ProcQ, RemovePackage, Mapp, Map14, Map6,**

Definition2, StrStr, ContextQ, Cnames, ToString1 }] Out[3346]= {Null**,** Null**,** Null**,** Null**,** Null**,** Null**,** Null**,** Null**,** Null**,** Null} In[3347]**:= Get["AVZ_Package.mx"]**
In[3348]**:= Definition[StrStr]**
Out[3348]= StrStr[x_]**:=** If[StringQ[x]**,** "\"" <> x<> "\""**,** ToString[x]] In[3349]**:= PackageQ["AvzAgnVsvArtKr`"]**
Out[3349]= False
In[3350]**:= DumpSave["AvzAgnVsvArtKr.mx", "AvzAgnVsvArtKr`"]** Out[3350]= {"AvzAgnVsvArtKr`"}
In[3351]**:= DumpSaveP["AvzAgnVsvArtKr.mx", "AvzAgnVsvArtKr`"]** Out[3351]= $Failed

The **DumpSaveP** function qua of the test for an admissibility of the second argument uses logical function whose call**PackageQ[*x*]** returns*True* if***x* –** a package containing global symbols, and*False* otherwise. Naturally, package without global symbols of any interest doesn't represent. Really, according to the system agreements the package has to define global symbols without that the package can't be considered as such. In this connection the function call**DumpSaveP[*f, x*],** where***x*** isn't a package, returns**$Failed,** allowing to process situations of this type very simply programmatically. So, despite a formal correctness of definition of packages without the global symbols or***mx*–**files without context, a testing of actual packages which are uploaded into the current session is necessary what a simple function does, whose call**Packages[]** returns the list of contexts of the actual packages that are loaded into the current session [33,48]. The next section considers some additional means for operating with the user packages.

8.3. Additional means of operating with the user packages in the*Mathematica*software

Means of the **Mathematica** for operating with datafiles can be subdivided into*two* groups conditionally**:** the means supporting the work with datafiles which are automatically recognized at the address to them, and the means supporting the work with any datafiles. This theme is quite extensive and is in more detail considered in [30-33,52,60,64], here some additional means of work with the datafiles containing the user's packages will be considered.

Since means of access to files of formats, even automatically recognized by the**Mathematica,** don't solve a number of important enough problems, the user is compelled to program own means on the basis of standard tools and perhaps with use of own means. Qua of an useful example we will give the **DefFromPackage** procedure whose call**DefFromPackage[*x*]** returns the***3*–** element list, whose first element is definition in string format of a symbol***x*** whose context is different from {***"Global'", "System'"***}, the second element defines its usage whereas the third element defines attributes of the symbol. At that, on the symbols associated with***2*** specified contexts, the procedure call returns only the list of their attributes. The fragment below represents source code of the**DefFromPackage** procedure with examples of its usage.

In[2742] **:= DefFromPackage[x_ /; SymbolQ[x]] := Module[{a = Context[x], b = "", c = "", p, d = ToString[x], k = 1, h}, If[MemberQ[{"Global\`", "System\`"}, a], Return[Attributes[x]], h = a <> d; ToExpression["Save[" <> ToString1[d] <> "," <>**

ToString1[h] <> "]"]]; For[k, k < Infinity, k++, c = Read[d, String]; If[c === " ", Break[], b = b <> c]]; p = StringReplace[RedSymbStr[b, " ", " "], h <> "\`"–> ""]; {c, k, b}= {"", 1, ""}; For[k, k < Infinity, k++, c = Read[d, String]; If[c === " " || c === EndOfFile, Break[], b = b <> If[StringTake[c, {–1,–1}] == "\", StringTake[c, {1,–2}], c]]]; DeleteFile[Close[d]]; {p, StringReplace[b, " /: " <> d–> ""], Attributes[x]}]
In[2743]**:= DefFromPackage[StrStr]**
Out[2743]= {StrStr[x_]:= If[StringQ[x], StringJoin["**"**", x, "**"**"], ToString[x]],

StrStr **::**usage= **"**The call StrStr[x] returns an expression x in string format if x is different from string**;** otherwise**,** the double string obtained from an expression x is returned**."**, {}}

In[2744] **:= DefFromPackage[AvzAgn]**

Out[2744]= {}
In[2745]:= **SetAttributes[Ian, {Listable, Protected}]; DefFromPackage[Ian]**
Out[2745]= {Listable, Protected}
In[2746]:= **DefFromPackage[Cos]**
Out[2746]= {Listable, NumericFunction, Protected}

The **DefFromPackage** procedure serves for obtaining of full information on a symbol***x*** whose definition is located in the user package uploaded into the current session. Unlike the standard functions**FilePrint** and**Definition** this procedure, first, doesn't print, but returns specified information completely available for the subsequent processing, and, secondly, this information is returned in an optimum format. At that, in a number of cases the output of definition of a symbol that is located in an active package by the standard means is accompanied with a context associated with the package that not only complicates its viewing, but also the subsequent processing. Result of the**DefFromPackage** call obviates this problem too. The algorithm realized by this procedure is based on an analysis of structure of a datafile received in result of saving of a context***"y'x",*** where***x* –** a symbol at the procedure call**DefFromPackage[*x*]** and***"y'"* –** a context, associated with the uploaded package containing the definition of symbol***x.*** In more detail the algorithm realized by the**DefFromPackage** procedure is seen from its source code.

As the second example developing the algorithm of the previous procedure in the light of application of functions of access it is possible to represent a rather useful**FullCalls** procedure whose the call**FullCalls[*x*]** returns the list whose first element is the context associated with a package uploaded into the current session whereas its other elements**–** the symbols of this package that are used by the user procedure or function***x*,** or nested list of sublists of this type at using by***x*** of symbols*(names of procedures/functions)* from several packages. The source code of the procedure along with typical examples of its usage are represented in the following fragment.

In[3435] **:= FullCalls[x_ /; ProcQ[x] || FunctionQ[x]] := Module[{a ={}, b, d, c = "::usage = ", k = 1}, Save[b = ToString[x], x]; For[k, k < Infinity, k++, d = Read[b, String]; If[d === EndOfFile, Break[], If[StringFreeQ[d, c], Continue[], AppendTo[a, StringSplit[StringTake[d, {1, Flatten[StringPosition[d, c]][[1]]–1}], " /: "][[1]]]]]]; a = Select[a, SymbolQ[#] &]; DeleteFile[Close[b]]; a = Map[{#, Context[#]}&, DeleteDuplicates[a]]; a = If[Length[a] == 1, a, Map[DeleteDuplicates, Map[Flatten, Gather[a, #1[[2]] === #2[[2]] &]]]]; {d, k}= {{}, 1}; While[k <= Length[a], b = Select[a[[k]], ContextQ[#] &]; c = Select[a[[k]], ! ContextQ[#] &]; AppendTo[d, Flatten[{b, Sort[c]}]]; k++]; d = MinusList[If[Length[d] == 1, Flatten[d], d], {ToString[x]}]; If[d == {Context[x]}, {}, d]]**

In[3436] **:= FullCalls[StrStr]**
Out[3436]= {}
In[3437]**:= G[x_] := StrStr[x] <> "RansIan50090"; FullCalls[G]** Out[3437]= {"AladjevProcedures`", "StrStr"}
In[3438]**:= F[x_/; IntegerQ[x], y_/; IntegerQ[y]] := x^2 + y^2; FullCalls[F]**
Out[3438]= {}
In[3439]**:= FullCalls[ProcQ]**

Out[3439]= {”AladjevProcedures`”, “BlockFuncModQ“, “ClearAllAttributes“,

“ Contexts1“, “Definition2“,”HeadPF“, “HowAct“, “ListStrToStr“, “Map3“, “Mapp“, “MinusList“, “PureDefinition“, “Sequences“, “StrDelEnds“, “SubsDel“, “SuffPref“, “SymbolQ“, “SysFuncQ“, “SystemQ“, “ToString1“, “UnevaluatedQ”}

In[3440] **:= FullCalls[Attribs]**
Out[3440]= {”AladjevProcedures`”, “Adrive“, “CopyDir“, “CopyFileToDir“, “DirQ“, “FileExistsQ1“, “HowAct“, “LoadExtProg“, “Map3“, “PathToFileQ“, “SearchFile“, “StandPath“, “StrDelEnds“, “StrStr“, “SuffPref“, “SymbolQ“, “ToString1”}
In[3441]**:= GS[x_ /; RuleQ[x], y_ /; StringQ[y]] := ArtKr[StringLength[StringReplace[y, x]], 500] + Vgs[StringLength[y], 90]; FullCalls[GS]** Out[3441]= {{”AladjevProcedures`”, “RuleQ”}, {”Kherson`”, “ArtKr“, “Vgs”}} In[3442]**:= GS[“Avz”–> “2015”, “AgnAvzVsvArtKr”]**
Out[3442]= 7604

Thus, the procedure call **FullCalls[*x*]** provides possibility of testing of the user procedure or function, different from standard means, regarding use by it of means whose definitions are in packages uploaded into the current session. In development of this procedure the**FullCalls1** procedure can be offered whose source code along with rather typical examples of its usage are represented by the following fragment.

In[2661] **:= FullCalls1[x_ /; ProcQ[x] || FunctionQ[x]] := Module[{a = {}, b, c = ””, d, k = 1, n, p}, Save[b = ToString[x], {x, c}]; For[k, k < Infinity, k++, d = Read[b, String];**

If[d === EndOfFile, Break[],
If[d != ” “, c = c <> d,
If[n = Flatten[StringPosition[c, ” := “]]; n != {}, If[Quiet[HeadingQ[p = StringTake[c, {1, n[[1]]–1}]]],

AppendTo[a, Quiet[HeadName[StringTake[c, {1, n[[1]]–1}]]]]]]; c = ””]]]; DeleteFile[Close[b]]; {b = FullCalls[x], Select[MinusList[a, {ToString[x]}], ! MemberQ[Flatten[b], #] &]}]

In[2662] **:= ArtKr[x_Integer, y_Integer] := Module[{}, N[Sqrt[x^2 + y^2]]]; Vgs[x_Integer, y_Integer] := N[Sin[x] + Cos[y]]; GS[x_ /; RuleQ[x], y_/; StringQ[y]] :=**

ArtKr[StringLength[StringReplace[y, x]], 90]+ Vgs[StringLength[y], 500];
In[2663]**:= FullCalls1[GS]**
Out[2663]= {{”AladjevProcedures`”, “RuleQ”}, {”ArtKr“, “ArtKr“, “Vgs”} In[2664]**:= FullCalls1[StrStr]**
Out[2664]= {{}, {}}
In[2665]**:= FullCalls1[ProcQ]**
Out[2665]= {{”AladjevProcedures`”, “BlockFuncModQ“,

“ ClearAllAttributes“, “Contexts1“, “Definition2“, “HeadPF“, “HowAct“, “ListStrToStr”,”Map3“, “Mapp“, “MinusList“, “PureDefinition“, “Sequences“, “StrDelEnds“, “SubsDel“, “SuffPref“, “SymbolQ“, “SysFuncQ“, “SystemQ“, “ToString1“, “UnevaluatedQ”}, {}}

The **FullCalls1** procedure tests procedure/function***x*** regarding use by it of both package means, and the other means, other than the standard means. In particular, the call**FullCalls1[*x*]** returns the nested list whose*first* element corresponds to result of the call**FullCalls[*x*]** while the*second* element defines the list of names of the means used by***x*,** excluding the means belonging to the uploaded user packages. Meanwhile, it must be kept in mind that both procedures process only the means used by***x*** which are determined by the mechanism of the delayed calculations. Spreading of these procedures onto the mechanism of immediate calculations of any special difficulties doesn**'**t cause, and such extension can present an useful enough exercitation to the interested reader. We proceeded from the fact that the definition of both the procedures, and the functions on a number of fairly significant reasons it is advisable to determine by the mechanism of delayed calculations. At that, both procedures**FullCalls** and**FullCalls1** are quite useful at programming a number of appendices. Right there quite pertinently to note that the**Save** function used in realization of the procedures**FullCalls** and**FullCalls1** can be quite useful for the organization of libraries of the user tools. Indeed, the call**Save[*f*, {*a*, *b*, …}]** saves in a datafile***f*** of the text format all definitions not only of objects with names {***a*, *b*, *c*, …**}, but also all definitions of means with which the specified objects are connected at all levels of their*structural* tree. At the same time, the function call writes into datafile in the***Append*** mode, leaving the datafile closed. Moreover, the created datafile is easily edited by simple text editors, allowing rather simply to create software for its editing *(deleting of objects*,*addition of objects*,*replacement of objects*,*etc.)*. For uploading of similar library into the current session the function call**Get[*f*]** is enough, having provided access to all means whose definitions were earlier saved in the datafile***f*.** The given question is considered rather in details in [30-33]. In a number of cases there is a need for uploading into the current session of the**Mathematica** system not entirely of a package, but only the separate means contained in it, for example, of a procedure/function, or their list. In the following fragment the procedure is represented, whose procedure call **ExtrOfMfile[*x*, *y*]** returns*Null*, i.e. nothing, uploading in the current session the definitions only of those means that are determined by argument***y*** and are located in a datafile***x*** of***m*–**format. At that, in case of existence in the***m*–** file of several means of the same name, the last is uploaded into the current session. While the call**ExtrOfMfile[*x*, *y*, *z*]** with the third optional argument ***z*** *–an undefinite variable–* in addition through***z*** returns the list of definitions of means***y*** which are located in the***m*–**file***x*.** In case of absence in a***m*–**file***x*** of means***y*** the procedure call returns**$Failed.** The next fragment represents source code of the**ExtrOfMfile** procedure along with examples of its usage.

```
In[2572]:= ExtrOfMfile[f_ /; FileExistsQ[f] && FileExtension[f] == “m”, s_ /;
StringQ[s] || ListQ[s], z___] :=

Module[ {Vsv, p = {}, v, m}, m = ReadFullFile[f]; If[StringFreeQ[m,
Map[“(*Begin["`” <> # <> “`"]*)” &, Map[ToString, s]]], $Failed, Vsv[x_, y_] :=
Module[{a = m, b = FromCharacterCode[17], c = FromCharacterCode[24], d =
“(*Begin["`” <> y <> “`"]*)”, h = “(*End[]*)”, g = {}, t}, a = StringReplace[a, h–> c];
If[StringFreeQ[a, d], $Failed, While[! StringFreeQ[a, d], a = StringReplace[a, d–> b,
1]; t = StringTake[SubStrSymbolParity1[a, b, c][[1]], {4,–4}]; t = StringReplace[t,
{“(*”–> ””, “*)”–> ””}]; AppendTo[g, t]; a = StringReplace[a, b–> ””, 1];
Continue[]]; {g, ToExpression[g[[–1]]]}]]; If[StringQ[s], v = Quiet[Check[Vsv[f, s]
```

[[1]], $Failed]], Map[{v = Quiet[Check[Vsv[f, #][[1]], $Failed]], AppendTo[p, v]}&, Map[ToString, s]]]; If[{z}!={}&& ! HowAct[z], z =If[StringQ[s], v, p]];]]

In[2573] := **ExtrOfMfile["C:\AVZ_Package\Kiev.m", "W"]** In[2574]:= **ExtrOfMfile["C:\AVZ_Package\Kiev.m", "W", w]** In[2575]:= {**W[73, 68, 90], w**} Out[2575]= {18053, {"W[x_Integer, y_Integer]:= x^2+ y^2",

" W[x_Integer, y_Integer, z_Integer]:= x^2+y^2+z^2", "W[x_Integer,y_Integer,z_Integer]:= x^3+y^3+z^3"}} In[2576]:= **ExtrOfMfile["C:/AVZ_Package/Kiev.m", {"W", "W1", "GS"}, w2]** In[2576]:= **w2**

Out[2576] = {{"W[x_Integer, y_Integer]:= x^2+ y^2", "W[x_Integer, y_Integer, z_Integer]:= x^2+ y^2+ z^2", "W[x_Integer, y_Integer, z_Integer]:= x^3+ y^3+ z^3"},

{ "W1[x_Integer, y_Integer]:= x*y+ W[x, y]"}, $Failed} In[2577]:= **ExtrOfMfile["C:/Temp/Kiev.m", {"AgnVsvArtKr", "Avz"}]** Out[2577]= $Failed
In[2578]:= **Remove[StrStr]**
In[2578]:= **Definition[StrStr]**
Out[2578]= Null
In[2579]:= {**ExtrOfMfile["c:/users/aladjev/mathematica/AVZ_Package.m",**

"StrStr", G], G}
Out[2579]= {Null, {"StrStr[x_]:= If[StringQ[x], "\""<>x<>"\"",

ToString[x]] "}}
In[2580]:= **Definition[StrStr]**
Out[2580]= StrStr[x_]:= If[StringQ[x], "\"" <> x<> "\"", ToString[x]]

It should be noted that this procedure can be quite useful in case of need of recovery in the current session of the damaged means without uploading of the user packages containing their definitions.

The **DefFromM** procedure directly adjoines to the**ExtrOfMfile** procedure, whose call**DefFromM[*x*, *y*]** returns definition of an object with a name***y*** that is located in a datafile***x*** of***m***–format with package while the procedure call **DefFromM[*x*,*y*, *z*],** where***z* –** an arbitrary expression, in addition evaluates this definition in the current session, making the object***y*** available. In order to simplification of algorithm of the**DefFromM** procedure the**SubListsMin** procedure is used, in general useful at operating with lists. The procedure call**SubListsMin[*L*,*x*, *y*, *t*]** returns the sublists of a list***L*** that are limited by elements {***x***,***y***} and have the minimum length; at***t* = "*r*"** selection is executed from left to right, and at***t* ="*l*"** from right to left. Whereas the procedure call **SubListsMin[*L*, *x*, *y*, *t*, *z*]** with optional*fifth* argument***z****-arbitrary expression*- returns sublists without the limiting elements {***x***, ***y***}. The following fragment represents source codes of both procedures with examples of their usage.

In[2742]:= **SubListsMin[L_/; ListQ[L], x_, y_, t_ /; MemberQ[{"r", "l"}, t], z___] := Module[{a, b, c, d = {}, k = 1, j}, {a, b}= Map[Flatten, Map3[Position, L, {x, y}]];**

If[a == {}|| b == {}|| a == {}&& b == {}|| L == {}, {}, b = Select[Map[If[If[t == "r", Greater, Less][#, a[[1]]], #] &, b], ! SameQ[#, Null] &]; For[k, k <= Length[a], k++, j = 1; While[j <= Length[b], If[If[t == "r", Greater, Less][b[[j]], a[[k]]], AppendTo[d,

If[t == "r", a[[k]] ;; b[[j]], b[[j]] ;; a[[k]]]]; Break[]]; j++]]; d = Sort[d, Part[#1, 2]–Part[#1, 1] <= Part[#2, 2]–Part[#2, 1] &]; d = Select[d, Part[#, 2]–Part[#, 1] == Part[d[[1]], 2]–Part[d[[1]], 1] &]; d = Map[L[[#]] &, d]; d = If[{**z**}**!= **{}**, Map[#[[2 ;;–2]] &, d], d]; If[Length[d] == 1, Flatten[d], d]]]**

:= SubListsMin[{**a, b, a, c, d, q, v, d, w, j, k, d, h, f, d, h**}**, a, h, "r", 90]**In[2743]
Out[2743]= {c, d, q, v, d, w, j, k, d}
In[2744]:= **SubListsMin[**{**h, g, a, b, h, a, c, d, a, q, h, v, w, a, j, k, d, h, f, d, h**}**,**

a, h, "r"] Out[2744]= {{a, b, h}, {a, q, h}}
In[2745]:= **SubListsMin[**{**h, g, a, b, h, a, c, d, a, q, h, v, w, j, k, d, h, f, d, h**}**,**

a, h, "r", 500] Out[2745]= {{b}, {q}}
In[2746]:= **SubListsMin[**{**h, g, a, b, h, a, c, d, a, q, h, v, w, j, k, d, h, f, d, h**}**,**

a, h, "l"] Out[2746]= {h, g, a}
In[2747]:= **SubListsMin[**{**h, g, a, b, h, a, c, d, a, q, h, v, w, j, k, d, h, f, d, h**}**,**

a, h, "l", 500] Out[2747]= {g}
In[2749]:= **DefFromM[x_ /; FileExistsQ[x] && FileExtension[x] == "m", y_ /; SymbolQ[y], z___] := Module[**{**a = ReadList[x, String], b, c, d**}**,**

{ b, c}**= **{**"(*Begin["`" <> ToString[y] <> "`"]*)", "(*End[]*)"**}**; d = StringJoin[Map[StringTake[#, **{**3,–3**}**] &, Flatten[SubListsMin[a, b, c, "r", 90]]]]; If[**{**z**}**!= **{}**, ToExpression[d]; d, d]]**

In[2750] **:= DefFromM["AVZ_Package.m", StrStr]; Definition[StrStr]** Out[2750]= StrStr[x_]**:=** If[StringQ[x]**,** "\"" <> x<> "\""**,** ToString[x]] Being based on the approach, used in the previous**ExtrOfMfile** procedure, and also on the mechanism of string patterns, we receive useful procedure which provides receiving of the list of means, whose definitions are located in the user package*(**m**–file)*. The procedure call**ContentOfMfile[*f*]** returns the list of names in string format of all means, whose definitions are located in a package*(**m**–file)* determined by argument***f*.** In absence in the***m***–file of definitions of tools in the*standard* package format the procedure call returns the empty list, i.e. {}. The following fragment represents source code of the **ContentOfMfile** procedure along with typical examples of its usage.

In[2830]:= **ContentOfMfile[f_ /; FileExistsQ[f] && FileExtension[f] ==**

"m"] := Module[{**b, a = ReadFullFile[f]**}**, b = StringSplit[a,** {**"(*", "*)"**}**]; b = Select[b, ! StringFreeQ[#,** {**"Begin["`", "`"]"**}**] && StringFreeQ[#, "BeginPackage["] &]; b = Flatten[Map[StringCases[#, ""`" ~~ __ ~~ "`""] &, b]]; b = DeleteDuplicates[Map[StringTake[#,** {**3,–3**}**] &, b]]; Sort[Select[b, StringFreeQ[#,** {**"=", ",", "`", "[", "]", "(", ")", "^", "^", ";", "{", "}", "\", "/"**}**] &]]]**

:= ContentOfMfile["C:\AVZ_Package\Kiev.m"] In[2841]
Out[2841]= {"W", "W1"}
In[2842]:= **ContentOfMfile["AVZ_Package.m"]**
Out[2842]= {"AcNb"**,** "ActBFMuserQ"**,** "ActCsProcFunc"**,** "ActRemObj"**,**

" ActUcontexts"**,** "AddMxFile"**,** "Adrive"**,** "Adrive1"**,** "Affiliate"**,** "Aobj"**,** "Aobj1"**,** "Args"**,** "Args1"**,** "ArgsBFM"**,** "ArgsTypes"**, …..,** "$InBlockMod"**,** "$Line1"**,**

"$Load$Files$", "$ProcName", "$ProcType", "$TestArgsTypes", "$TypeProc", "$UserContexts"}

In[2843]:= **Length[%]**
Out[2843]= 683

The previous **ContentOfMfile** procedure can be simplified and reduced to a function, using**SubsString** function providing the allocation of substrings from a string on condition of*satisfaction* of the allocated substrings to the set conditions. The procedure call**SubsString[*s*, {*a, b, c, d, …*}]** returns the list of substrings of a string***s*** that are limited by substrings {***a, b, c, d, …***} whereas the procedure call**SubsString [*s*, {*a, b, c, d, e, …*},*p*]** with the third optional argument***p*** *–a pure function in short format–* returns the list of substrings of a string***s*** that are limited by substrings {***a,b, c, d, …***}, meeting the condition determined by a pure function***p.*** While the procedure call**SubsString[*s*, {*a, b, c, d, …*},*p*]** with the***3****rd* optional argument***p*** *–an arbitrary expression which different from pure function–* returns a list of substrings limited by substrings {***a, b, c, d, …***}, with removed prefixes and suffixes {***a, b, c, d, …***}[[***1***]] and {***a, b, c, d, …***}[[***-1***]] accordingly. In absence in a string***s*** of at least one of substrings {***a, b, c, d, …***} the procedure call returns the empty list. Using the**SubsString** procedure, it is rather simple to modify the**ContentOfMfile** procedure in the form of**ContentOfMfile1** function whose source code with source code of**SubsString** procedure along with certain examples of their typical usage the following fragment represents.

In[3215] := **SubsString[s_/; StringQ[s], y_/; ListQ[y], pf___] := Module[{b, c, a = "", k = 1}, If[Set[c, Length[y]] < 2, s, b = Map[ToString1, y]; While[k <= c–1, a = a <> b[[k]] <> "~~ Shortest[__] ~~ "; k++]; a = a <> b[[–1]]; b = StringCases[s, ToExpression[a]]; If[{pf}!= {}&& PureFuncQ[pf], Select[b, pf], If[{pf}!= {}, Map[StringTake[#, {StringLength[y[[1]]] + 1, –StringLength[y[[–1]]]–1}] &, b], Select[b, StringQ[#] &]]]]]**

In[3216] := **SubsString["adfghbffgxbavzgagngbArtggbKgrg", {"b","g"}, StringFreeQ[#, "f"] &]**
Out[3216]= {"bavzg", "bArtg", "bKg"}
In[3217]:= **SubsString["adfghbffgxbavzgagngbArtgbKrg", {"b", "g"}]**
Out[3217]= {"bffg", "bavzg", "bArtg", "bKrg"}
In[3218]:= **SubsString["abcxxxxx42345abcyyyyy42345", {"ab", "42"}, 90]**
Out[3218]= {"cxxxxx", "cyyyyy"}

In[3227]:= **ContentOfMfile1[f_ /; FileExistsQ[f] && FileExtension[f] == "m"] := Sort[DeleteDuplicates[Select[Map[StringTake[#, {9,–4}] &,**

SubsString[ReadFullFile[f], {"Begin["`", "`"]"}]], StringFreeQ[#, {"=", ",", "`", "[", "]", "(", ")", "^", ";", "{", "}", "\", "/"}] &]]] In[3228]:= **ContentOfMfile1["C:\Temp\AVZ_Package\Kherson.m"]** Out[3228]= {"W", "W1"}
In[3229]:= **ContentOfMfile1["AVZ_Package.m"]**
Out[3229]= {"AcNb", "ActBFMuserQ", "ActCsProcFunc", "ActRemObj",

" ActUcontexts", "AddMxFile", "Adrive", "Adrive1", "Affiliate", "Aobj", "Aobj1", "Args", "Args1", "ArgsBFM", "ArgsTypes", **….., "$InBlockMod", "$Line1", "$Load$Files$", "$ProcName", "$ProcType", "$TestArgsTypes", "$TypeProc",**

“$UserContexts”}

In[3230]**:= Length[%]**
Out[3230]= 683

In general it should be noted that the **Mathematica** system posesses a rather developed mechanism of*string patterns* which allows to program developed means of processing of various string structures.

The two procedures below are quite useful at manipulations with a package that is located in a***mx***–file. So, the procedure call**ContextMXfile[*x*]** returns the context associated with the user’s package which is located in a***mx***–file ***x***. Meanwhile, uploading of the***mx***–file into the current session isn’t made. The**MxToTxt** procedure allows***2 ÷ 4*** actual arguments. The procedure call **MxToTxt[*x, y*]** returns*Null***,** i.e. nothing, saving in a datafile***y*** of***txt***–format and in the current session all definitions of a package which is located in a ***mx***–file. At that, all definitions of the file***x*** are saved in an optimum format *(without the context associated with package)*. If the call**MxToTxt[*x, y, z*],** since the third argument, contains optional argument***“Del”*,** the package***x*** isn’t loaded into the current session, otherwise all its definitions are saved in the current session in optimum format. If at the procedure call the arguments, starting with the third, contain an undefinite variable, through it the list of all objects whose definitions are located in a file***x*** with the user package is returned. The following fragment represents source codes of the mentioned procedures together with them associated, and the examples of their usage.

In[2825]**:= ContextMXfile[x_ /; FileExistsQ[x] && FileExtension[x] == “mx”] := Module[**{**a, c,**

Flatten[Map7[Range, Sequences, {{**48, 57**}**,** {**65, 90**}**,** {**96, 122**}}**]]**}**, a =BinaryReadList[x]; a = a[[1;; If[Length[a] >=500, 500, Length[a]]]]; c = Flatten[Map3[PosSubList, a,** {{**67, 79, 78, 84**}**,** {**69, 78, 68, 67, 79, 78, 84**}}**]]; If[Length[c] < 5, $Failed, FromCharacterCode[Select[a[[c[[2]] +1 ;; c[[5]]–1]], MemberQ[b, #1] &]]]]**

In[2826]**:= ContextMXfile[“F:\AVZ_Package\AVZ_Package.mx”]** Out[2826]= **“**AladjevProcedures`**”**

In[2827] **:= ContextFromFile[x_/; StringQ[x]] := If[Quiet[FileExistsQ[x]] && MemberQ[**{**“m”, “nb”, “mx”, “cdf”**}**, FileExtension[x]], Quiet[ToExpression[StringJoin[“Context”, ToUpperCase[If[FileExtension[x] == “cdf”, “nb”, FileExtension[x]]], “file[“, ToString1[x], “]”]]], $Failed]**

In[2828] **:= Map[ContextFromFile,** {**“E:\Temp/Kherson.m”, “Package.nb”, “C:\Users\Aladjev\Mathematica\AVZ_Package.mx”**}**]**
Out[2828]= {**”**Kherson`**”, “**AladjevProcedures`**”, “**AladjevProcedures`**”**}

In[2829]**:= MxToTxt[x_ /; FileExistsQ[x] && FileExtension[x] == “mx”, y_ /; StringQ[y], z___] := Module[**{**b, c, a = ContextMXfile[x]**}**,**

LoadMyPackage[x, a]; b = CNames[a]; Map[{**Write[y, Definition[#]], Write[y]**}**&, b]; Close[y]; If[MemberQ[**{**z**}**, “Del”], RemovePackage[a]]; c = Select[**{**z**}**, ! HowAct[#] && ! SameQ[#, “Del”] &]; If[c != **{}**, ToExpression[ToString[c[[1]]] <> “=” <> ToString[b]]];]**

```
BeginPackage["Kherson`"]
Gs::usage = "Function Gs[x, y] := 73*x^2 + 68*y + 47 + S[x, y]." G::usage =
"Function G[x, y] := N[Sin[x] + Cos[y]] + S[x, y]." V::usage = "Function S[x_, y_] :=
x^2 + y^2."
Begin["`Private`"]
V[x_, y_] := x^2 + y^2
Gs[x_ /; IntegerQ[x], y_ /; IntegerQ[y]] := 73*x^2 + 68*y + 47 + V[x, y] G[x_ /;
IntegerQ[x], y_ /; IntegerQ[y]] := N[Sin[x] + Cos[y]] + V[x, y]

End[]
EndPackage[]
In[2830]:= $Packages

Out[2830] = {"AladjevProcedures`", "ResourceLocator`", "PacletManager`", "System`",
"Global`"}
In[2831]:= ContextMXfile["Kherson.mx"]
Out[2831]= "Kherson`"
In[2832]:= MxToTxt["Kherson.mx", "Kherson.txt"]
In[2833]:= $Packages
Out[2833]= {"Kherson`", "AladjevProcedures`", "GetFEKernelInit`",
"TemplatingLoader`", "ResourceLocator`", "PacletManager`", "System`", "Global`"}
In[2834]:= MxToTxt["Kherson.mx", "Kherson.txt", g]; g
Out[2834]= {G, Gs, V}
In[2835]:= $Packages
Out[2835]= {"Kherson`", "AladjevProcedures`", "GetFEKernelInit`",
"TemplatingLoader`", "ResourceLocator`", "PacletManager`", "System`", "Global`"}
In[2836]:= MxToTxt["Kherson.mx", "Kherson.txt", "Del"]
In[2837]:= $Packages
Out[2837]= {"AladjevProcedures`", "GetFEKernelInit`",
"TemplatingLoader`", "ResourceLocator`", "PacletManager`", "System`", "Global`"}
In[2838]:= MxToTxt["Kherson.mx", "Kherson.txt", t, "Del"]; t
Out[2838]= {G, Gs, V}
In[2839]:= $Packages
Out[2839]= {"AladjevProcedures`", "GetFEKernelInit`",
"TemplatingLoader`", "ResourceLocator`", "PacletManager`", "System`", "Global`"}
```

The **MxToTxt** procedure has***2*** rather useful modifications**MxToTxt1** and **MxToTxt2** with which it is possible to familiarize in [30-33,48]. In particular, on the basis of the procedures**MxToTxt ÷ MxToTxt2** it is possible to create quite effective and simple libraries of the user means with system of their maintaining. The similar organization is rather habitual for the users having experience in traditional programming systems.

The next equivalent procedures **ContextFromMx** and**ContextFromMx1** use different algorithms**;** their calls on a***mx*–**file return a context ascribed to the user package, at a context absence**$Failed** is returned. The fragment below presents source codes of these procedures and an auxiliary function along with typical examples of their usage. The function call**StringFreeQ2[*x*, {*a1, a2, a3, …*}]** returns*True* if all substrings {***a1, a2, a3, …***} are absent in a string ***x*,** and*False* otherwise.

In[2770]:= **ContextFromMx[x_ /; FileExistsQ[x] && FileExtension[x] ==**

"mx"] := Module[{d = Map[FromCharacterCode, Range[2, 27]], a = StringJoin[Select[Characters[ReadString[x]], SymbolQ[#] || IntegerQ[#] || # == "`" &]], b}, If[StringFreeQ2[a, {"CONT", "ENDCONT"}], $Failed, b = StringCases1[a, {"CONT", "ENDCONT"}, "___"]; If[b == {}, $Failed, StringReplace[b, Flatten[{GenRules[d, ""], "ENDCONT"–> "", "CONT"–> ""}]][[1]]]]]

In[2771]:= **ContextFromMx["c:\users/aladjev/mathematica/Tallinn.mx"]** Out[2771]= "Grodno`"
In[2772]:= **ContextFromMx1[x_ /; FileExistsQ[x] && FileExtension[x] ==**

"mx"] := Module[{d = Map[FromCharacterCode, Range[2, 27]], a = StringJoin[Select[Characters[ReadString[x]], SymbolQ[#] || IntegerQ[#] || # == "`" &]], b}, If[StringFreeQ2[a, {"CONT", "ENDCONT"}] || StringCases1[a, {"CONT", "ENDCONT"}, "___"] == {}, $Failed, b = StringPosition[a, {"CONT", "ENDCONT"}]; If[b =={}, $Failed, StringReplace[StringTake[a, {b[[1]][[1]], b[[2]][[2]]}], Flatten[{GenRules[d, ""], "ENDCONT"–> "", "CONT"–> ""}]]]]]

In[2773]:= **ContextFromMx1["c:/users/aladjev/mathematica/Tallinn.mx"]**
Out[2773]= "Grodno`"
:= StringFreeQ2[x_ /; StringQ[x], y_ /; StringQ[y] || ListQ[y] &&In[2774]
DeleteDuplicates[Map[StringQ[#] &, y]] == {True}] := ! MemberQ[Map[StringFreeQ[x, #] &, Flatten[{y}]], False]

In[2775] **:= StringFreeQ2["12tvArthsnm3p45k6r78hKr9", {"a", "b", "c", "d"}]**
Out[2775]= True
It should be noted that both procedures operates on platforms*WindowsXP Professional* and*Windows7Professional*. Moreover performance of procedures is higher if they are applied to a***mx***–file created on the current platform. In view of distinctions of the***mx***–files created on different platforms there is a natural expediency of creation of the means testing any***mx***–file regarding a platform in which it was created in virtue of the**DumpSave** function. The following**TypeWinMx** procedure is one of such means. The procedure call **TypeWinMx[*x*]** in string format returns the type of operating platform on which a***mx***–file***x*** was created**;** correct result is returned for case of*Windows* platform, while on other platforms**$Failed** is returned. This is conditioned by lack of a possibility to carry out debugging on other platforms. The next fragment represent source code of the procedure with examples of its use.

In[2785] **:= TypeWinMx[x_ /; FileExistsQ[x] && FileExtension[x] == "mx"]:= Module[{a, b, c, d}, If[StringFreeQ[$OperatingSystem, "Windows"], $Failed, a = StringJoin[Select[Characters[ReadString[x]], SymbolQ[#] ||Quiet[IntegerQ[ToExpression[#]]] ||# == "–" &]]; d = Map[FromCharacterCode, Range[2, 27]]; b = StringPosition[a, {"CONT", "ENDCONT"}]; If[b[[1]][[2]] == b[[2]][[2]],**

c = StringCases1[a, {"Windows", "ENDCONT"}, "___"], b = StringPosition[a, {"Windows", "CONT"}]; c = StringTake[a, {b[[1]][[1]], b[[2]][[2]]}]]; c = StringReplace[c, Flatten[{GenRules[d, ""],

"ENDCONT"–> "", "CONT"–> ""}]]; If[ListQ[c], c[[1]], c]]]

In[2786] **:= TypeWinMx["ProcQ.mx"]**
Out[2786]= "Windows-x86-64"
In[2787]**:= TypeWinMx["AVZ_Package_1.mx"]**
Out[2787]= "Windows"
In[2788]**:= TypeWinMx["AVZ_Package.mx"]**
Out[2788]= "Windows-x86-64"

The call **DumpSave[*x, y*]** returns the list of contexts***y*** of objects or objects with definitions that were ostensibly unloaded into a***mx***–file***x*** irrespective of existence of*definitions* for these objects or their*contexts* in the list defined by**$ContextPath** variable, without allowing to make program processing of results of**DumpSave** calls. At that, the result of the**DumpSave** call can't be tested programmatically and elimination of this situation is promoted by a procedure, whose successful call **DumpSave1[*x, y*]** returns the nested list whose*first* element defines the path to a file***x*** of***mx***–format*(if necessary,the **"mx"**extension is ascribed to the datafile)* while the*second* element defines the list of objects and/or contexts from the list defined by***y*** whose definitions are unloaded into the datafile***x.*** In the absence of objects*(the certain symbols and/or contexts existing in the list defined by the**$ContextPath**variable)* which were defined by argument***y,*** the**DumpSave1** call returns**$Failed.** The next fragment represents source codes of the procedure along with other means, useful at processing of datafiles of***mx***–format, and contexts of symbols.

In[3120]**:= DumpSave1[x_, y_] := Module[{a, b, c},**

If[StringQ[x], If[FileExtension[x] == "mx", c = x, c = x <> ".mx"]; a = Flatten[{y}]; b = Select[a, (ContextQ[#] && MemberQ[$ContextPath, #]) || ! MemberQ[{"", "Null"}, Quiet[ToString[Definition[#]]]] &]; If[b != {}, {c, Flatten[DumpSave[c, b]]}, $Failed], $Failed]]

In[3121]**:= DumpSave1["AVZ_Package42.mx", {"Art`", "Kr`", GS}]** Out[3121]= $Failed

In[3124] **:= ReplaceSubLists[x_/; ListQ[x], y_ /; RuleQ[y] || ListRulesQ[y]] := Module[{a, f, d = FromCharacterCode[2015]}, f[z_/; ListQ[z]] := StringJoin[Map[ToString1[#] <> d &, z]]; a = Map[f[Flatten[{#[[1]]}]]–> f[Flatten[{#[[2]]}]] &, Flatten[{y}]]; ToExpression[StringSplit[StringReplace[f[x], a], d]]]**

In[3125] **:= ReplaceSubLists[{a, b, c, "d", m, x, b, c}, {{b, c}–> {x, y}, a–> {m, n}, "d"–> "90"}]**
Out[3125]= {m, n, x, y, "90", m, x, x, y}

In[3128]**:= SubsList[x_/; ListQ[x], y_, z_] := Module[{b, c,**
a = FromCharacterCode[2015]}, b = StringJoin[Map[ToString1[#] <> a &, x]]; c = Map[StringJoin[Map[ToString1[#] <> a &, Flatten[{#1}]]] &, {y, z}]; c = ToExpression[StringSplit[SubsString[b, {c[[1]], c[[2]]}], a]]; If[Length[c] == 1, c[[1]], c]] In[3129]**:= SubsList[{a, b, c, d, x, y, x, b, c, n, a + b, x, y, z}, {b, c}, {x, y}]**
Out[3129]= {{b, c, d, x, y}, {b, c, n, a+ b, x, y}}
In[3146]**:= ContextToSymbol[x_/; SymbolQ[x], y_ /; ContextQ[y], z_ /; StringQ[z]] := Module[{b = Flatten[{PureDefinition[x]}], a = If[FileExtension[z] == "mx", z, z <> ".mx"]}, If[b === {$Failed}, $Failed, AppendTo[$ContextPath, y]; Quiet[ToExpression[Map[y <> # &, b]]]; {a, DumpSave[a, y]}]]**

In[3147] **:= Art[x_] := Module[{a = 6}, x + a];**
ContextToSymbol[Art, "Veeroja`", "AgnAvz"]
Out[3147]= {"AgnAvz.mx", {"Veeroja`"}}

In[3150] **:= ContextRepMx[x_ /; FileExistsQ[x]&& FileExtension[x] =="mx", y_ /; ContextQ[y]] := Module[{a = ContextMXfile[x], b, c, d, h, n, m, f}, a = If[SameQ[a, $Failed], "None", a]; b = ReadList[x, Byte]; c = Map[ToCharacterCode, {"Windows", "ENDCONT"}]; f = ToString[Unique[]] <> ".mx"; ContextToSymbol[d, y, f]; h = ReadList[f, Byte]; n = SubsList[b, c[[1]], c[[2]]]; m = SubsList[h, c[[1]], c[[2]]]; DeleteFile[f]; h = ReplaceSubLists[b, n–> m]; f = FileNameSplit[x]; f = FileNameJoin[AppendTo[f[[1 ;;–2]], "$" <> f[[–1]]]]; BinaryWrite[f, h]; Close[f]; {f, a, y}]**

In[3151]**:= ContextRepMx["Kherson.mx", "Grodno`"]**
Out[3151]= {**"**$Kherson.mx**", "**Kherson`**", "**Grodno`"}
In[3154]**:= ContextSymbol[x_/; SymbolQ[x]] := Select[Map[If[MemberQ[CNames[#], ToString[x]] || MemberQ[CNames[#], # <> ToString[x]], #] &, DeleteDuplicates[$ContextPath]], ! SameQ[#1, Null] &]**

In[3155] **:= Map[ContextSymbol, {G, Gs, ProcQ, Sin}]**
Out[3155]= {{"Kherson`"}, {"Grodno`"}, {"AladjevProcedures`"}, {"System`"}} So, the previous fragment represents as the main, and supportive means of processing of***mx***–files and contexts. The procedure call**ReplaceSubLists[*x, y*]** returns the result of replacement of elements*(including adjacent)* of a list***x*** on the basis of a rule or list of rules***y;*** moreover, lists can be as parts of rules. Whereas the procedure call**SubsList[*x, y, z*]** returns the list of sublists of the elements of a list***x*** that are limited by elements {***y,z***}**;** qua of elements {***y, z***} can be lists too. If any of elements {***y, z***} doesn't belong***x,*** the procedure call returns the*empty* list, i.e. {}. The presented procedures**ReplaceSubLists** and **SubsList** along with processing of lists are of interest for assignment to***mx***– files of a context in its absence.

Whereas the procedure call **ContextToSymbol[*x,y, z*]** returns the list of the format {***z,*** {***y***}}, ascribing to a certain symbol***x*** a context***y*** with saving of its definition in a***mx***–file***z.*** In particular, the given means is quite useful in the case of necessity of saving of objects in***mx***–files with a context. On the basis of three procedures**ReplaceSubLists, SubsList** and**ContextToSymbol** the procedure which provides replacement of contexts in***mx***–files without their uploading into the current session has been created. So, the procedure call **ContextRepMx[*x,y*]** provides replacement of the context of a***mx***–file***x*** by a new context***y,*** returning the list of the format {***File, h, y***} where***File*** **–** the file with result of such replacement,***h*** **–** an old context or***"None"*** **–** if it wasn't, and***y*** **–** a new context. Whereas the function call**ContextSymbol[*x*]** returns the context associated with a symbol***x.***

At calculation of definition of a symbol ***x*** in the current session the symbol will be associated with the***"Global'"*** context that remains at its unloading into***mx***file by means of the**DumpSave** function. While in some cases there is a need of saving of symbols in***mx***-files with other contexts. The procedure **DumpSave2** solves the given problem whose call**DumpSave2[*f,x,y*]** returns nothing, unloading into a***mx***–file***f*** the definition of a symbol or their list***x*** that have context***"Global'"*** with***y*** context. The fragment below represents source code of the**DumpSave2** procedure along with examples of its usage.

In[3222] **:= DumpSave2[x_ /; FileExtension[x]== "mx", y_ /; SymbolQ[y]|| ListQ[y] && DeleteDuplicates[Map[SymbolQ[#] &, y]]=={True}, z_ /; ContextQ[z]] := Module[{b, c, a = Flatten[Select[Map[PureDefinition[#] &, Flatten[{y}]], ! SameQ[#, $Failed] &]]}, Map[ToExpression[z <> #] &, a]; AppendTo[$ContextPath, z]; c = Map[z <> ToString[#] &, Flatten[{y}]]; AppendTo[c, $ContextPath]; DumpSave[x, c];]**

In[3223] **:= Agn[x_] := x; Agn[x_, y_] := x + y; Agn[x_Integer] := x + 500** In[3224]**:= Avz[x_] := x^2; Avz[x_, y_] := 90*(x+y); Avz[x_Integer] := x+500** In[3225]**:= Map[ContextSymbol,** {**Agn, Avz**}**]**
Out[3225]= {{"Global`"}, {"Global`"}}
In[3226]**:= DumpSave2["Tallinn.mx",** {**Agn, Avz**}**, "Rans`"]**
In[3227]**:= Clear[Avz, Agn]; Map[PureDefinition,** {**Agn, Avz**}**]** Out[3227]= {$Failed, $Failed}
In[3228]**:= Get["Tallinn.mx"]; PureDefinition[Rans`Agn]**
Out[3228]= {"Rans`Agn[x_Integer]:= x+ 500**"**, **"**Rans`Agn[x_]:= x**"**,

" Rans`Agn[x_, y_]:= x+ y"}
In[3229]**:= Ian`Agn[x_, y_] := x + y; PrependTo[$ContextPath, "Ian`"]** Out[3229]= {"Ian`**"**, **"**AladjevProcedures`**"**, **"**TemplatingLoader`**"**,

" PacletManager`**"**, **"**System`", "Global`**"**, **"**Rans`"}
In[3230]**:= ContextSymbol[Agn]**
Out[3230]= {"Ian`**"**, **"**Rans`"}
In[3231]**:= DumpSave["Tampere.mx", "Ian`"];**
In[3232]**:= $ContextPath = MinusList[$ContextPath,** {**"Ian`"**}**];**

PureDefinition[Agn]
Out[3232]= $Failed
In[3233]**:= Get["Tampere.mx"]; CNames["Ian`"]**
Out[3233]= {"Agn"}
In[3234]**:= ContextMXfile["Tampere.mx"]**
Out[3234]= **"**Ian`**"**
In[3235]**:= PureDefinition[Agn]**
Out[3235]= **"**Agn[x_, y_]:= x+ y**"**

The previous fragment is completed by examples illustrating the principle of saving of objects, whose definitions are evaluated in the current session, in***mx*–**files with the given context. This principle was used at programming of the procedures endowing a symbol by a context.
As it was noted earlier, the objects of the same name have various headings therefore in certain cases arises a question of their more exact identification. The next procedure provides one of such approaches, trying to associate the components composing such objects with the contexts ascribed to them. At the heart of the procedure algorithm lies a principle of creation for separate components of an object of the same name of packages in***m*–**files with the unique contexts ascribed to them. Then, having removed an object***x*** of the same name from the current session, by means of uploading of these***m***files into the current session we have opportunity of access to components of the object***x*** of the same name through a construction of the***"Context'x"*** format. The fragment below represents

souce code of the**DiffContexts** procedure along with typical examples of its usage.

```
In[2630] := DiffContexts[x_ /; SymbolQ[x] && ! UnevaluatedQ[HeadPF, x], y___] :=
Module[{a = {"(*BeginPackage[".12`"]*)", "(*.0f::usage=""" <> "*)",
"(*Begin["`.06`"]*)", "(*.04*)", "(*End[]*)", "(*EndPackage[]*)"},

b = Map[FromCharacterCode, {18, 15, 6, 4}], c = Definition2[x][[1 ;;–2]], d, h =
ToString[x], k = 1, j, t = {}, p, f = {}, z}, If[Length[c] < 2, Context[x], z = HeadPF[x];
Clear[x]; For[k, k <= Length[c], k++, d = {}; For[j = 1, j <= Length[a], j++,
AppendTo[d, StringReplace[a[[j]], {b[[1]]–> h <> ToString[k], b[[2]]–> h, b[[3]]–> h,
b[[4]]–> c[[k]]}]]]; AppendTo[t, p = h <> ToString[k] <> ".m"]; AppendTo[f, {h <>
ToString[k] <> "`", z[[k]]}]; Map[{BinaryWrite[p, ToCharacterCode[#][[3 ;;–3]]],
BinaryWrite[p, {32, 10}]}&, d]; Close[p]; Get[p]]; If[{y}!= {}, Map[DeleteFile, t],
Null]; Reverse[f]]]

In[2631] := T[x_] := x; T[x_, y_] := x*y; T[x_, y_, z_] := x*y*z
In[2632]:= DiffContexts[T]
Out[2632]= {{"T3`", "T[x_, y_, z_]"}, {"T2`", "T[x_, y_]"}, {"T1`", "T[x_]"}}
In[2633]:= Definition["T1`T"]
Out[2633]= T1`T[T1`T`x_]:= T1`T`x
In[2634]:= Definition["T2`T"]

Out[2634] = T2`T[T2`T`x_, T2`T`y_]:= T2`T`x*T2`T`y
In[2635]:= Definition["T3`T"]
Out[2635]= T[T3`T`x_, T3`T`y_, T3`T`z_]:= T3`T`x*T3`T`y*T3`T`z In[2636]:=
Definition[T]
Out[2636]= T[T3`T`x_, T3`T`y_, T3`T`z_]:= T3`T`x*T3`T`y*T3`T`z In[2637]:=
$Packages
Out[2637]= {"T3`", "T2`", "T1`", "AladjevProcedures`", "GetFEKernelInit`",

" TemplatingLoader`", "ResourceLocator`", "PacletManager`", "System`", "Global`"}
In[2638]:= DiffContexts[T, 500]
Out[2639]= {"T3`", "T2`", "T1`"}
In[2639]:= Definition["T1`T"]
Out[2639]= T1`T[T1`T`x_]:= T1`T`x
In[2640]:= FileExistsQ["T1.m"]
Out[2640]= False
In[2641]:= T3`T[73, 68, 48]
Out[2641]= 238 272
```

The procedure call **DiffContexts[*x*]** returns the nested list of***ListList***–type whose sublists by the*first* element define***context*** while the second element define***heading*** of a certain component of an object of the same name***x*** in the format {{***"xn'","cn'"***}, **…,** {***"x2'", "c2'"***}, {***"x1'", "c1'"***}} whose order is defined by order of the contexts in the list defined by the**$Packages** variable, where ***n –*** number of components of the object of the same name***x.*** Moreover, the datafiles***"xj.m"*** with the packages with components definitions composing the object of the same name***x*** remain in the current directory of the session ***(j=1..n)***. At the same time the procedure call**DiffContexts[*x, y*]** with the***2nd*** argument***y*** *–an arbitrary expression–* returns the above result, removing the intermediate***m***-files.

Whereas on*x* objects different from objects of the same name the procedure call**DiffContexts[*x*]** returns the context of an object***x*.** A certain interest is represented by the**NamesCS** procedure whose the call **NamesCS[*P, Pr, Pobj*]** returns*Null,* i.e. nothing while thru three arguments ***P, Pr, Pobj* –***undefinite variables***–** are respectively returned the list of contexts corresponding to the packages uploaded into the current session, the list of the user procedures, whose definitions are activated in the***Input***paragraph of the current session, and the nested list, whose sublists in the main have various length and are structurally formatted as follows**:**

–*the first element of a sublist defines the context corresponding to a package which was uploaded in the current session of the**Mathematica**system at the time of the **NamesCS**procedure call**;***

–*all subsequent elements of this sublist define objects of this package which in the current session of the**Mathematica**system were made active**.***

The following fragment represent source code of the**NamesCS** procedure along with a typical example of its usage.

```
In[2593]:= NamesCS[P_ /; ! HowAct[P], Pr_ /; ! HowAct[Pr],
Pobj_ /; ! HowAct[Pobj]] :=

Module[ {b = Contexts[], c = $Packages, d, k = 1, p, n, m, h, a =
Quiet[Select[Map[ToExpression, Names["`*"]], ProcQ[#] &]]}, {P, Pr}= {c, a}; c =
Map[List, c]; For[k, k <= Length[b], k++, For[p = 1, p <= Length[c], p++, n = b[[k]];
m = c[[p]][[1]]; If[n === m, Null, If[SuffPref[n, m, 1], d = StringReplace[n, b–> ""];
If[d == "", Null, c[[p]] = Append[c[[p]], ToExpression[StringTake[StringReplace[n,
b–> ""], {1,–2}]]]]], Continue[]]]]; c = Map[DeleteDuplicates, c]; For[k = 1, k <=
Length[c], k++, h = c[[k]]; If[Length[h == 1], h = Null, h = Select[h, StringQ[#] ||
ToString[Quiet[DefFunc[#]]] != "Null" &]]]; Pobj := Select[c, Length[#] > 1 && ! #
=== Null &]; Pobj = Mapp[Select, Pobj, If[! StringQ[#], True, If[StringTake[#,–1] ==
"`", True, False]] &]; ]

In[2594] := NamesCS[P, Pr, Pobj]
In[2595]:= {P, Pr}
Out[2595]= {{"Grodno`", "Tallinn`", "Kiev1`", "Kiev`", "AladjevProcedures`",

" GetFEKernelInit`", "TemplatingLoader`", "ResourceLocator`", "PacletManager`",
"System`", "Global`"}, {}}
In[2596]:= Pobj
Out[2596]= {{"Kherson`", Ga, Gs, Private, Vgs},
{"AladjevProcedures`", ActBFMuserQ, ActRemObj, ..., {"ResourceLocator`",
Private},

{ "PacletManager`", Collection`Private, Private, ...}, {"QuantityUnits`", Private},
{"WebServices`", Information},
{"System`", BesselParamDerivativesDump, BinaryReadDump}}
```

Moreover, the list returned through ***Pobj*–**argument contains only sublists, whose corresponding packages have objects that have been activated in the current**Mathematica** session.

While the call **Npackage[*x*]** of very simple function returns the list of names in string format of all objects whose definitions are located in a*x* package activated in the current session. In case of inactivity in the current session of the package*x* or in case of its absence the function call**Npackage[*x*]** returns **$Failed.** So, the following fragment represents source code of the**Npackage** function along with typical examples of its usage.

In[2684]:= **Npackage[x_/; StringQ[x]] := If[MemberQ[Contexts1[], x], Sort[Select[Names[x <> "*"], StringTake[#,–1] != "$" && ToString[Definition[#]] != "Null" &]], $Failed]**

In[2685] := **Npackage["AladjevProcedures`"]**
Out[2685]= {"AcNb", "ActBFMuserQ", "ActCsProcFunc", "ActRemObj", "AddMxFile", "Adrive", "Adrive1", "Affiliate", "Aobj", …, "$CallProc", "$InBlockMod", "$Line1", "$ProcName", "$ProcType", "$TestArgsTypes", "$TypeProc", "$UserContexts"}
In[2686]:= **Npackage["Tallinn`"]**
Out[2686]= $Failed

The **ContOfContex** procedure also is represented as a rather interesting tool whose call**ContOfContex[*x*]** returns the nested two-element list whose first element defines the sublist of all names in string format of means of the user package with a context*x* whose*definitions* in the current session are returned by the**Definition** function with the*x* context included in them whereas the second element defines the sublist of all names in string format of means of the package with the*x* context whose definitions in the current session are returned by the**Definition** function without context*x.* The fragment below represents source code of the**ContOfContex** procedure with an example of its use concerning the context***"AladjevProcedures'"*** that is associated with the***AVZ_Package*** package [48]. At the end of the fragment the*length* of both sublists of the returned result is calculated along with random inspection by means of the**Definition** function of definitions of means from both sublists. From the received estimation follows, the length of the first sublist of means of the above package whose*definitions* in the current session are returned by the**Definition** function along with the context is significantly longer.

In[2705]:= **ContOfContext[x_ /; ContextQ[x]] := Module[{b = {}, c = {}, h, a = Select[CNames[x], # != "a" &], k = 1},**

If[a == {}, $Failed, While[k <= Length[a], h = a[[k]]; If[StringFreeQ[StringReplace[ToString[Definition4[h]], "\n \n"–> ""], x <> h <> "`"], AppendTo[c, h], AppendTo[b, h]]; k++]; {b, c}]]

In[2706]:= **ContOfContext["AladjevProcedures`"]**
Out[2706]= {{"ActBFMuserQ", "ActRemObj", "AddMxFile", "Adrive1",

" Affiliate", "Aobj", "Aobj1", "Args", "Args1", "ArgsBFM", ……}, {"AcNb", "ActCsProcFunc", "Adrive", "Attributes1", "Avg", "BlockQ", "BlockQ1", "CALL",…,"$Load$Files$", "$ProcName", "$ProcType","$TestArgsTypes","$TypeProc","$UserContexts"}} In[2707]:= **Map[Length, %]**
Out[2707]= {425, 259}
In[2708]:= **Definition["DirName"]**
Out[2708]= DirName[AladjevProcedures`DirName`F_ /;

StringQ[AladjevProcedures`DirName`F]]**:=** If[DirQ[AladjevProcedures`DirName`F]**,** **"**None**"**, If[! FileExistsQ1[AladjevProcedures`DirName`F]**,** $Failed, Quiet[Check[FileNameJoin[FileNameSplit[AladjevProcedures`DirName`F][[1**;–**2]]]**,** **"**None**"**]]]] In[2709]**:= Definition["StrStr"]**
Out[2709]= StrStr[x_]**:=** If[StringQ[x]**,** StringJoin[**"**"**"**, x, **"**"**"**]**,** ToString[x]] In[2710]**:= ContOfContext["AladjevProceduresAndFunctions`"]** Out[2710]= $Failed

In[2716]**:= LoadPackage[x_ /; FileExistsQ[x]&& FileExtension[x] == "mx"]:= Module[**{**a**}**, Quiet[ToExpression["Off[shdw::Symbol]"]; Get[x]; a = ToExpression["Packages[][[1]]"];**

ToExpression["LoadMyPackage[" <> **"**"**"** <> **x** <> **"**"**"** <> **","** <> **"**"**"** <> **a** <> **"**"**"** <> **"]"]; ToExpression["On[shdw::Symbol]"]]]**

In[2717] **:= LoadPackage["C:\Users\Mathematica\AVZ_Package.mx"]** In[2718]**:= Definition["DirName"]**
Out[2718]= DirName[F_ /; StringQ[F]]**:=** If[DirQ[F]**,** **"**None**"**,

If[! FileExistsQ1[F]**,** $Failed**,** Quiet[Check[FileNameJoin[FileNameSplit[F][[1**;–**2]]]**,** **"**None**"**]]]]

On inactive contexts ***x*** the procedure calls**ContOfContext[*x*]** return**$Failed** while in other cases the procedure call is returned unevaluated. Qua of one of possible appendices of the given procedure it is possible to note problems that deal with source codes of software of the user packages. For*elimination* of similar distinction the**LoadPackage** procedure completing the previous fragment can be used. The procedure call**LoadPackage[*x*]** returns*Null***,** i.e. nothing, loading the user package contained in a datafile***x*** of the***mx***format into the current session of the**Mathematica** with activation of*all* definitions which contain in it in a mode similar to the mode of the***Input***paragraph of the**Mathematica** system.

Qua of useful addition to the **ContOfContex** procedure, the**NamesContext** procedure can be quite considered,whose call**NamesContext[*x*]** returns the list of names in string format of program objects of the current session that are associated with a context***x*.** In case of absence of this context the empty list, i.e. {} is returned. If the***x*** value is different from a context the procedure call is returned unevaluated. The following fragment represents source code of the**NamesContext** procedure along with typical examples of its usage.

In[2840]**:= NamesContext[x_ /; ContextQ[x]] := Module[**{**b, c =** {}**, k = 1, h, a = Names[x** <> **"*"]**}**,**

While[k <= Length[a], b = a[[k]]; h = ToString[ToExpression["Definition[" <> **b** <> **"]"]]; If[h != "Null" && h != "Attributes["** <> **b** <> **"]=**{**Temporary**}**" && ! SuffPref[b, "a$", 1], AppendTo[c, a[[k]]]]; k++]; c]** In[2841]**:= NamesContext["AladjevProcedures`"]**
Out[2841]= {**"**AcNb**"**, **"**ActBFMuserQ**"**, **"**ActCsProcFunc**"**, **"**ActRemObj**"**,

" AddMxFile**"**, **"**Adrive**"**, **"**Adrive1**"**, **"**Affiliate**"**, **"**Aobj**"**, **…**, **"**$CallProc**"**, **"**$InBlockMod**"**, **"**$Line1**"**, **"**$ProcName**"**, **"**$ProcType**"**, **"**$TestArgsTypes**"**, **"**$TypeProc**"**, **"**$UserContexts**"**}

In[2842] **:= Length[%]**
Out[2842]= 684

In[2843]:= **NamesContext["Global`"]**
Out[2843]= {"Art", "G", "Gal", "Kr", "Global`LoadPackage", "NamesContext"}
In[2844]:= **Length[%]**
Out[2844]= 6
In[2845]:= **NamesContext["System`"]**
Out[2845]= {"\[FormalA]", "\[FormalB]", "\[FormalC]", "\[FormalD]",

"\ [FormalE]", "\[FormalF]", "\[FormalG]", "\[FormalH]", … , "$UserName", "$Version", "$VersionNumber", "$WolframID", "$WolframUUID", "\[SystemsModelDelay]"}

In[2846] := **Length[%]**
Out[2846]= 5167
In[2847]:= **NamesContext["Tallinn`"]**
Out[2847]= {}

The procedure call **Contexts1[]** that is a simple modification of the**Contexts** function which provides testing of an arbitrary string for admissibility qua of a*syntactically* correct context returns the list of contexts corresponding to packages whose components have been activated in the current session. The following fragment represents source code of the**Contexts1** procedure with a typical example of its usage.

In[2920]:= **Contexts1[] := Module[{a = {}, b = Contexts[], c, k = 1}, For[k, k <= Length[b], k++, c = b[[k]]; If[Length[DeleteDuplicates[Flatten[StringPosition[c, "`"]]]] == 1 && StringTake[c, {–1,–1}] == "`", AppendTo[a, c], Next[]]]; a]**

In[2921] := **Contexts1[]**
Out[2921]= {"AladjevProcedures`", "Algebra`", "AlphaIntegration`", ……} In[2922]:= **Length[%]**
Out[2922]= 184
In some cases exists the problem of definition of the***m***–files containing the definition of some object active in the current session. The given problem is successfully solved by the procedure whose call**FindFileObject[*x*]** returns the list of datafiles containing definition of an object***x*,** including the usage**;** in the absence of such***m***–files the procedure call returns the empty list, i.e. {}. The procedure call**FindFileObject[*x*, *y*, *z*, …]** with optional arguments {***y*, *z*, …**} qua of which the names in string format of devices of direct access are defined, provides search of***m***-files on the specified devices instead of search in all file system of the computer by the procedure call with one argument. The next fragment represents source code of the**FindFileObject** procedure along with some typical examples of its usage.

In[4363]:= **FindFileObject[x_ /; ! SameQ[ToString[DefOpt[ToString[x]]],**

"Null"], y___] := Module[{b = {}, c = "", s = {}, d, k = 1, a = If[{y}== {}, Adrive[], {y}], f = "ArtKr", h = "(*Begin["`" <> ToString[x] <> "`"]*)", p = "(*" <> ToString[x] <> "::usage=", t}, While[k <= Length[a], Run["Dir ", a[[k]] <> ":\", " /B/S/L > "<>f]; While[! SameQ[c, "EndOfFile"], c = ToString[Read[f, String]]; If[StringTake[c, {–2,–1}] == ".m", AppendTo[b, c]]; Continue[]]; Quiet[Close[f]]; c = ""; k++]; k = 1; While[k <= Length[b], If[Select[ReadList[b[[k]], String], ! StringFreeQ[#, h] && StringFreeQ[#, p] &] != {}, AppendTo[s, b[[k]]]]; k++];

{**DeleteFile[f], s**}**[[2]]]**

In[4364] **:= FindFileObject[ProcQ, "D"]**
Out[4364]= {"d:\grgu_books\avz_package\avz_package.m",
"d:\temp\aladjev\documents\avz_package.m",
"d:\temp\aladjev\mathematica\avz_package.m"}
In[4365]**:= Mapp[FindFileObject,** {**Mapp, AvzAgn**}**]**
Out[4365]= {{"c:\grgu_books\avz_package\avz_package.m",
"c:\users\aladjev\documents\avz_package.m",
"c:\users\aladjev\mathematica\avz_package.m", "e:\avz_package\avz_package.m"},
FindFileObject[AvzAgn]}
So, for identification of means of the user package whose definitions in the current session contain contextual references, the following procedure can be used, whose call**DefWithContext[*x*]** returns the ***returns the***element nested list**:** its *first* element defines the list of names of means of the package loaded from a***m***file***x*** whose definitions don't contain contextual references whereas the *second* element– the list of names of means of the package whose definitions contain the contextual references. The following fragment represents source code of the procedure and examples of its usage prior to the procedure call **ReloadPackage1** and after it that is rather illustratively.

In[2982] **:= DefWithContext[x_ /; FileExistsQ[x] && FileExtension[x] == "m"] :=
Module[**{**a = ContextMfile[x], b, c =**{}**, d =**{}}**, b = CNames[a];
Map[If[StringFreeQ[Definition4[#], a<># <>"`"], AppendTo[c, #],**

AppendTo[d, #]] &, b]; {**c, d**}**]**

In[2983] **:= DefWithContext["C:\Mathematica\AVZ_Package.m"]** Out[2983]=
{{"AcNb", "ActCsProcFunc", "Adrive", "Attributes1", "Avg", "BlockQ", "BlockQ1",
"CALL", "CDir", "ClearAllAttributes", …, "$CallProc", "$InBlockMod", "$Line1",
"$Load$Files$", "$ProcType", "$TestArgsTypes", "$UserContexts"}, {"ActBFMuserQ",
"ActRemObj", "AddMxFile", "Adrive1", "Affiliate", "Aobj", "Aobj1", "Args",
"Args1", "ArgsBFM", …., "WhatType", "WhichN", "XOR1", "$ProcName",
"$TypeProc"}}
In[2984]**:= Map[Length, %]**
Out[2984]= {258, 425}
In[2985]**:= ReloadPackage1["C:\Mathematica\AVZ_Package.m"]**
In[2986]**:= d = DefWithContext["C:\Mathematica\AVZ_Package.m"];**
In[2987]**:= Map[Length, d]**
Out[2987]= {683, 0}

From the given fragment follows that more than ***62.2%*** of definitions of the means of our***AVZ_Package*** package uploaded into the current session, that are received by means of the function call**Definition[*x*]** will contain context references of the format***"AladjevProcedures'x'".***

At uploading of the user package into the current session its context will be located in the list determined by the**$Packages** variable while at attempting to receive definitions of its means by means of the**Definition** function some such definitions will contain the context associated with this package. First of all, such definitions are much less readable, but not this most important. For software that is based on optimum format and using similar definitions, in the process of working with them the erroneous situations are possible as it was already noted above. For the purpose of receiving definitions of tools of the user package in*optimal* format the**LoadMyPackage** procedure can be used. The procedure call**LoadMyPackage[*x, y*]** at the very beginning of the current session of the**Mathematica** returns*Null***,** i.e.*nothing***,** loading the user package***x*** with***y*** context ascribed to it, with the subsequent reevaluation of definitions of its means, providing the optimal format of these definitions.

In[2593]**:= LoadMyPackage[x_ /; FileExistsQ[x] && FileExtension[x] == "mx", y_] := Module[{a, Cn, Ts, k = 1},**

Ts[g_] := Module[{p = "$Art26Kr18$.txt", b = "", c, d, v = 1}, Write[p, g]; Close[p]; While[v < Infinity, c = Read[p, String]; If[SameQ[c, EndOfFile], Close[p]; DeleteFile[p]; Return[b], b = b <> c]; Continue[]]]; Cn[t_] := Module[{s = Names[StringJoin[t, "*"]], b},

b = Select[s, Quiet[ToString[Definition[ToString[#1]]]] != "Null" &]]; Quiet[Get[x]]; a = Cn[y]; While[k <= Length[a], Quiet[ToExpression[StringReplace[StringReplace[Ts[ToExpression["Definition[" <> a[[k]] <> "]"]], y–> ""], a[[k]] <> "`"–> ""]]]; k++]]

In[2594] **:= LoadMyPackage["AVZ_Package.mx", "AladjevProcedures`"]** In[2595]**:= Definition["ContextQ"]**
Out[2595]= ContextQ[x_]**:=** StringQ[x]**&&** StringLength[x]> 1**&&** Quiet[

SymbolQ[Symbol[StringTake[x**,** {1**,**–2}]]]]**&&**StringTake[x**,** {–1**,**–1}]==**"`"**

The previous fragment adduces source code of **LoadMyPackage** procedure with an example of its application. Similar approach is recommended to be used at uploading of the user package, saved in a datafile of***mx***–format, for elimination of the specified undesirable moments and for simplification of programming with use of its means, and also for extension of the system on the basis of its means. Furthermore, the procedure call**LoadMyPackage[*x*,*y*]** with the noted purposes can be executed and in the presence of the loaded user package***x*** with***y*** context. So, saving of a package in the***mx***–file with its subsequent uploading in each new session by the**Get** function, providing access to all package means with receiving their definitions in an optimized format*(in the above–mentioned sense)* is the most effective.

In the course of operating in the current session with means of an uploaded package*(from**m**–file)* situations when certain of its activated means for one reason or another are removed from the current session or are distorted are quite real. For their restoration the**ReloadPackage** procedure can be used.

In[2992] **:= ReloadPackage[x_ /; FileExistsQ[x] && FileExtension[x]=="m", y___List, t___] := Module[{a = NamesMPackage[x], b = ContextMfile[x], c ="$Art26Kr18$.txt", p, k = 1, d = If[{y}!= {}, ToExpression[Map14[StringJoin, Map[ToString, y], "[", 90]], {}]}, Put[c]; While[k <=Length[a], p =a[[k]]; PutAppend[StringReplace[ToString1[ToExpression["Definition[" <> p <> "]"]], b <> p <> "`"–> ""], c]; k++]; If[d == {}, ToExpression["Clear[" <> StringTake[ToString[a], {2,–2}] <> "]"], Null]; While[b != "EndOfFile", b =**

ToString[Read[c]]; If[b === "EndOfFile", Break[]];

If[d == {}, Quiet[ToExpression[b]]; Continue[], If[If[{t}== {}, MemberQ, ! MemberQ]

[d, StringTake[b, {1, Quiet[StringPosition[b, "[", 1][[1]][[1]]]}]], Quiet[ToExpression[b]]; Break[], Continue[]]]]; Close[c]; DeleteFile[c]]

In[2993] **:= ReloadPackage["C:\Mathematica\AVZ_Package.m"]** In[2994]**:= Definition[StrStr]**
Out[2994]= StrStr[x_]**:=** If[StringQ[x]**,** "\"" <> x<> "\""**,** ToString[x]] In[2995]**:= Clear[StrSts]; Definition[StrStr]**
Out[2995]= Null
In[2996]**:= ReloadPackage["C:\Mathematica\AVZ_Package.m"]** In[2994]**:= Definition[StrStr]**
Out[2994]= StrStr[x_]**:=** If[StringQ[x]**,** "\"" <> x<> "\""**,** ToString[x]] The successful procedure call**ReloadPackage[*x*]** returns nothing, providing in the current session the activation of all means of a package that is located in a***m*–**file***x*** as though their definitions were calculated in an input stream. If the call**ReloadPackage[*x,y*]** contains the second optional***y***-argument qua of which the list of names is used, the reboot is made only for the package means with the given names. At the same time the call**ReloadPackage[*x,y,t*]** in addition with the***3rd*** optional argument where***t*** *–an arbitrary expression,* also returns nothing, providing reboot in the current session of all means of the package***x*,** excluding only means with the names given in the list***y*.** The previous fragment represents source code of the**ReloadPackage** procedure with typical examples of its usage. In particular, it is illustrated that reboot of a package provides more compact output of definitions of the means that are contained in it, i.e. the output of definitions is made in a so-called*optimal* format*(without contexts)***.** The following fragment represents source code of the**ReloadPackage1** procedure, functionally equivalent to**ReloadPackage** procedure, along with typical examples of its usage.
In[4436]**:= ReloadPackage1[x_ /; FileExistsQ[x]&& FileExtension[x]== "m",**

y_: 0, t_: 0] := Module[{a = NamesMPackage[x], b = ReadFullFile[x], c, d = Map[ToString, Flatten[{y}]]}, c = Flatten[Map[SubsString[b, {"*)(*Begin["`" <> # <> "`"]*)(*", "*)(*End[]*)"}, 90] &, a]]; c = Map[StringReplace[#, "*)(*"–> ""] &, c]; Map[If[d == {"0"}, Quiet[ToExpression[#]], If[ListQ[y], If[{t}== {0}, If[MemberQ[d,

StringTake[#, Flatten[StringPosition[#, {"[", " :=", "="}]][[1]]–1]], ToExpression[#], If[! MemberQ[d, StringTake[#, Flatten[StringPosition[#, {"[", " :=", "="}]][[1]]–1]],

ToExpression[#]]]]]] &, c];] In[4437]**:= Map[Clear, {StrStr, Map2}]**
Out[4437]= {Null**,** Null}
In[4438]**:= Definition[StrStr]**
Out[4438]= Null
In[4439]**:= Definition[Map2]**
Out[4439]= Null

In[4440] **:= ReloadPackage1["C:\Mathematica\AVZ_Package.m", {StrStr, Map2}]**
In[4441]**:= Definition[StrStr]**

Out[4441]= StrStr[x_]**:=** If[StringQ[x]**,** **“"”** <> x<> **“"”,** ToString[x]]
In[4442]**:= Definition[Map2]**
Out[4442]=Map2[F**_** /**;** SymbolQ[F]**,** c**_** /**;** ListQ[c]**,** d**_** /**;** ListQ[d]]**:=**
(Symbol[ToString[F]][#1**,** Sequences[d]]**&**) /**@** c

The successful procedure call **ReloadPackage1[*x*]** returns*nothing***,** providing in the current session the activation of all means of a package that is located in a***m*–**file***x*** as though their definitions were calculated in an input stream. If the call**ReloadPackage1[*x*,*y*]** contains the*second* optional***y***-argument qua of which the list of names is used, the reboot is made only for the package means with the given names. Furthermore, the call**ReloadPackage1[*x*,*y*, *t*]** in addition with the***3****rd* optional argument where***t –****an arbitrary expression***,** also returns nothing, providing reboot in the current session of all means of the package***x*,** excluding only means with the names given in the list***y*.** At that, similar to**ReloadPackage** procedure the**ReloadPackage1** procedure, in particular, also provides output of definitions in the*optimal* format in the above sense. The given modification is of interest from the standpoint of the approaches used in it. Such approach allows to get rid of contextual links in definitions of the functions/procedures loaded into the current session from the user package. At that, with methods of uploading of the user packages into the current session it is possible to familiarize enough in details in [33].

As it was noted earlier, in the result of uploading into the current session of the user package from a file of format {***m*, *nb***} with its subsequent activation an essential part of definitions of its means received by the call of standard **Definition** function will include contextual links of the***“Context’x’”*** format, where***x* –** a name of means and***“Context”* –** a context ascribed to the given package. Means of identification of those objects of the user package whose definitions have contextual references are presented above. However these means suppose that the analyzed package is*activated* in the current session. Whereas the next procedure provides the similar analysis of an unuploaded package located in a datafile of***mx*–**format. The fragment below represents source code of the**MxPackNames** procedure with an example of its usage. The procedure call**MxPackNames[*x*]** returns the list of names of objects in string format of a***nb*–**file***y*** that is analog of a***mx*–**file***x*,** whose definitions in case of uploading of the datafile***y*** into the current session with subsequent activation the system**Definition** function will return with contextual links of the above–mentioned format.

In[3235] **:= MxPackNames[x_ /; FileExistsQ[x] && FileExtension[x] == “mx”] := Module[**{**b, c, d, g =** {}**, k, j, a = FromCharacterCode[Select[ToCharacterCode[ReadFullFile[x]], # > 31 &]]**}**, b = StringPosition[a,** {**“CONT”, “ENDCONT”**}**][[1 ;; 2]]; b = StringTake[a,** {**b[[1]][[2]] + 2, b[[2]][[1]]–1**}**]; b = Map[#[[2]] + 1 &, StringPosition[a, b][[2 ;;–1]]]; For[k = 1, k <= Length[b], k++, c = ””;**

For[j = b[[k]], j < Infinity, j++, d = StringTake[a, {**j, j**}**]; If[d == “`”, Break[], c = c <> d]]; AppendTo[g, c]]; Sort[g][[2 ;;–1]]]** In[3236]:= **MxPackNames[“C:\Mathematica\AVZ_Package.mx”]** Out[3236]= {”ActBFMuserQ**“,** “ActRemObj**“,** “AddMxFile“**,** “Adrive1**“,**

“ Affiliate**“,** “Aobj**“,** “Aobj1**“,** “Args**“,** “Args1**“,** “ArgsBFM**“, … ,**

===
"VarExch1", "Ver", "VizContentsNB", "VizContext", "WhatObj", "WhatType", "WhichN", "XOR1", "$ProcName", "$TypeProc"} In[3237]**:= Length[%]**
Out[3237]= 427
In[3238]**:= N[427*100/Length[CNames["AladjevProcedures`"]], 3]** Out[3238]= 62.3

Examples of the previous fragment once again confirm that the quantity of means of our***AVZ_Package*** package, uploaded into the current session from the ***nb***–file, whose definitions received by the**Definition** function contain contextual references, more than***62%.***

The question of obtaining the list of *names* of objects whose definitions with their usages are located in a package being in a datafile of format {***m, nb***} is represented interesting enough. At that, it is supposed that uploading of a package into the current session isn't obligatory. Such problem is solved by quite useful procedure, whose call**PackNames[*x*]** returns the list of names of the above objects in a package, being in a datafile***x*** of format {***m, nb***}. The next fragment represents source code of the**PackNames** procedure with an example of its application to the***AVZ_Package*** package which is located in *ASCII* datafiles***"AVZ_Package.m"*** and***"AVZ_Package.nb"*** [48].

In[2872] **:= PackNames[x_ /; FileExistsQ[x] && MemberQ[{"m", "nb"}, FileExtension[x]]] := Module[{a = ReadFullFile[x], b, c ={}, d = "", k = 1, j, h}, If[FileExtension[x] == "m",**

a = Select[DeleteDuplicates[Map[StringTake[#, {5,–1}] &, SubsString1[a, {"*)(*", "::usage="}, StringFreeQ[#, " "] &, 0]]], # != "" &]; Sort[Select[a, StringFreeQ[#, "(*"] &]], b = Quiet[SubsString1[a, {"RowBox[{ RowBox[{ RowBox[{", "\"::\"", "\"usage\"}]"}, StringQ[#] &, 0]]; b = Map[StringTake[#, {2,–8}] &, b]; Sort[Map[If[SymbolQ[#], #] &, b]]]]]

In[2873] **:= PackNames["C:\Mathemstica\AVZ_Package.m"]** Out[2873]= {"AcNb", "ActBFMuserQ", "ActCsProcFunc", "ActRemObj", "Adrive", "Adrive1", "Affiliate", "Aobj", "Aobj1", "Args", "ArgsBFM", "ArgsTypes", "Arity", "Arity1", "ArityBFM", **…..**}
In[2874]**:= Length[%]**
Out[2874]= 657
In[2875]**:= PackNames["C:\Mathematica\AVZ_Package.nb"]**
Out[2875]= {"AcNb", "ActBFMuserQ", "ActCsProcFunc", "ActRemObj", "Adrive", "Adrive1", "Affiliate", "Aobj", "Aobj1", "Args", "ArgsBFM", "ArgsTypes", "Arity", "Arity1", "ArityBFM", **…..**}
In[2876]**:= Length[%]**
Out[2876]= 677

It should be noted that the algorithm of **PackNames** procedure significantly uses the**SubsString1** procedure that is the**SubsString** procedure extension, being of interest in programming of the tasks connected with processing of strings. The procedure call**SubsString1[*s,y,f,t*]** returns the list of substrings of a string***s*** that are limited by substrings of a list***y;*** at that, if a testing pure function acts as argument***f,*** the returned list will contain only the substrings satisfying this test. Moreover, at***t* = *1*** the returned substrings are limited to ultra substrings of the list*y* whereas at***t* = *0*** substrings are returned without the limiting ultra substrings of the list***y.*** At last, in the presence of the fifth optional argument***r*** *–an arbitrary expression–* search of substrings in a string ***s*** is done from right to left, that as a whole simplifies algorithms of search of the required substrings. The following fragment represents source code of the**SubsString1** procedure along with some typical examples of its usage.

```
In[2770] := SubsString1[s_/; StringQ[s], y_/; ListQ[y], pf_/; PureFuncQ[pf], t_ /;
MemberQ[{0, 1}, t], r___] := Module[{c, h, a = ””, b = Map[ToString1, y], d = s, k =
1}, If[Set[c, Length[y]] < 2, s,

If[ {r}!= {}, b = Map[StringReverse, Reverse[b]]; d = StringReverse[s]]]; While[k <=
c–1, a = a <> b[[k]] <> “~~ Shortest[__] ~~ “; k++]; a = a <> b[[–1]]; h =
StringCases[d, ToExpression[a]]; If[t == 0, h=Map[StringTake[#,
{StringLength[b[[1]]]–1,
–StringLength[b[[–1]]]+1}] &, h]]; If[PureFuncQ[pf], h = Select[h, pf]];

If[{r}!= {}, Reverse[Map[StringReverse, h]], h]]

In[2771] := SubsString1[“12345#xyzttmnptttabc::usage=45678”, {“#”, “::usage=4”},
0, 0]
Out[2771]= {”xyzttmnptttabc”}
In[2772]:= SubsString1[“2345#xaybz::usage=5612345#xm90nyz::usage=
590#AvzAgn::usage=500”, {“#”, “::usage=”}, 0, 0]
Out[2772]= {”xaybz“, “xm90nyz“, “AvzAgn”}
In[2773]:= SubsString1[“12345#xyz::usage=45612345#x90yz::usage=500#
Avz::usage=590”, {“#”, “::usage=”}, 0, 1]
Out[2773]= {“#xyz::usage=”, “#x90yz::usage=”, “#Avz::usage=”}
In[2774]:= SubsString1[“12345#xyz::usage=45612345#x590yz::usage=500
#Avz::usage=590”, {“#”, “::usage=”}, LetterQ[#] &, 0]
Out[2774]= {”xyz“, “Avz”}
```

Here, in connection with the aforesaid it is quite appropriate to raise a quite important question concerning the global variables defined by a procedure. According to agreements of procedural programming, a variable defined in a procedure qua of*global* variable is visible outside of the procedure, i.e. can change own value both in the procedure, and outside of it, more precisely, field of its definition is the current session as a whole. In principle, the given agreement is fair and for the current session of the**Mathematica** system, but with very essential stipulations that are discussed in [30,33] with interesting enough examples. If a certain procedure defining global variables has been activated in the***Input***–stream, the above agreement is valid. Meanwhile, if such procedure is located in a datafile of format {***m***|***nb***}, then the*subsequent* uploading of such datafile into the current session makes active all means contained in the datafile, making them available, however the mechanism of global variables as a whole doesn’t work. In our work [33] an approach eliminating defects of the mechanism of global variables is represented.

For providing the mechanism of global variables *(including)*, a rather useful **LoadNameFromM** procedure was created whose call**LoadNameFromM[*f, n*]** provides uploading and activation in the current session of a procedure***n*** or their list saved in a datafile***f*** of the***m***–format with a package.

In[2588]:= **LoadNameFromM[F_ /; FileExistsQ[F] && FileExtension[F] ==**

"m" && StringTake[ToString[ContextFromFile[F]], –1] == "`", p_ /; StringQ[p] || ListStringQ[p]] := Module[{a =ReadFullFile[F], b ={}, c ="*)(*End[]*)", d, h = Flatten[{p}]}, b =Map[SubsString[a, {"(*Begin["`" <> # <> "`"]*)(*", c}, 90] &, h]; b = If[Length[b] == 1, Flatten[b], Map[#[[1]] &, b]]; Map[ToExpression, Map[StringReplace[#, "*)(*"–> " "] &, b]];]

In[2589] **:= LoadNameFromM["C:\Temp\AVZ_Package.m", "StrStr"]** In[2590]:= **Definition[StrStr]**
Out[2590]= StrStr[x_]**:=** If[StringQ[x]**,** "\"" <> x<> "\""**,** ToString[x]]

The previous fragment represents source code of the given procedure with an example of its usage. This procedure in a certain relation is adjoined also the next**ExtrPackName** procedure. The algorithm of the procedure is based on analysis of internal structure of a file of***m***–format with the user package. The successful procedure call**ExtrPackName[*f*,*w*]** returns*Null***,** i.e. nothing, with simultaneous return of the evaluated definition of an object***w*** which is contained in a***m***file***f*** with the user package, making the definition available in the current session. If the format of a datafile***f*** is other than***m***-format, the procedure call returns**$Failed,** whereas in absence in a file***f*** of the requested object***w*** the procedure call**ExtrPackName[*f*,*w*]** returns the corresponding message. The fragment below represents source code of the**ExtrPackName** procedure along with some typical examples of its usage.

In[2883] **:= ExtrPackName[F_ /; StringQ[F], N_ /; StringQ[N]] := Module[{a, b, c, d, Art, Kr}, If[FileExistsQ[F] && FileExtension[F] == "m" && StringTake[ToString[ContextFromFile[F]],–1] == "`", a = OpenRead[F], Return[$Failed]]; If[Read[a, String] != "(* ::Package:: *)", Close[a]; $Failed, {c, d}= {"", StringReplace["(*Begin["`Z`"]*)", "Z"–> N]}]; Label[Art]; b = Read[a, String]; If[b === EndOfFile, Close[a]; Return["Definition of " <> N <> " is absent in file <" <> F <> ">"], Null]; If[b != d, Goto[Art], Label[Kr]; b = StringTake[Read[a, String], {3,–3}]; c = c <> b <> " "; If[b == "End[]", Close[a]; Return[ToExpression[StringTake[c, {1,–8}]]], Goto[Kr]]]]**

In[2884] **:= ExtrPackName["F:\Mathematica\AVZ_Package.m", "Df"]** In[2885]**:= ExtrPackName["F:\Mathematica\AVZ_Package.m", "Subs"]** In[2886]**:= ExtrPackName["F:\Mathematica\AVZ_Package.m", "ArtKr"]** Out[2886]= **"**Definition of ArtKr is absent in file<F**:**\Mathematica\

AVZ _Package**.**m>**"**
In[2887]**:= ExtrPackName["C:\Temp\AVZ_Package_6.m", "ProcQ"]** Out[2887]= **$**Failed
In[2888]**:= Df[(Sin[1/x^2] + Cos[1/x^2])/x^2, 1/x^2]**
Out[2888]= x^2 (– (–1+ x^2) Cos[1/x^2]– (1+ x^2) Sin[1/x^2]) In[2889]**:= Subs[(Sin[1/x^2] + Cos[1/x^2])/x^2, 1/x^2, h]**
Out[2889]= (Cos[h]+ Sin[h])/h
In[2890]**:= ExtrPackName["C:\Temp\Tallinn.m", "Gs"]; Definition[Gs]** Out[2890]= Gs[x**_ /;** IntegerQ[x]**,** y**_ /;** IntegerQ[y]]**:=** x^2+ y^2

The given procedure provides activation in the current session of a concrete function or procedure which is located in a*m*–file without uploading of the datafile completely. By functionality the given procedure is crossed with the **LoadNameFromM** procedure considered above, however possesses certain additional useful opportunities.
As a rule, enough many of the user packages contain in own structure the variables of several types which appear at their uploading into the current session of the system. For definition of such variables the procedure can be used, whose call **UserPackTempVars[*x*]** returns the three–element nested list where the*first* sublist determines the*undefinite* variables associated with the package defined by a context***x*,** the second sublist defines the*temporary* variables associated with the package and having names of format**"Name$"** while the*third* sublist defines symbols of the format**"Name$Integer"** that in the current session aren't distinguished as symbols. The following fragment represents source code of the given procedure with an example of its usage.

In[2684]:= **UserPackTempVars[x_ /; ContextQ[x]] := Module[{a = {}, p, b = {}, d = {}, c = Names[x <> "*"], h = {}},**

Quiet[Map[{p = Definition2[#], If[UnevaluatedQ[Definition2, #], AppendTo[d, #], If[p[[2]] == {}&& p[[1]] == "Undefined", AppendTo[a, #], If[p[[2]] == {Temporary}, AppendTo[b, #], 6]]]}&, c]]; Map[{p = Flatten[StringPosition[#, "$"]], If[p[[–1]] == StringLength[#], AppendTo[a, #], If[IntegerQ[ToExpression[StringTake[#, {p[[–1]] + 1, StringLength[#]}]]], AppendTo[h, #]]]}&, b]; {d, a, h}]

In[2685]:= **UserPackTempVars["AladjevProcedures`"]**
Out[2685]= {{"a", "b", "c", "h", "k", "p", "S", "x", "y"}, {"a$", "b$", "c$", "d$", "h$", "k$", "Op$", "p$", "S$", "x$", "y$"},
{"a$30755", "b$30755", "c$30755", "d$30755", "p$30755"}}

In[2695] **:= $UserContexts :=**
Select[Map[If[Flatten[UserPackTempVars[#][[2 ;; 3]]] != {}, #] &, Select[$Packages, ! MemberQ[{"Global`", "System`"}, #] &]],

! SameQ[#, Null] &] In[2696]:= **$UserContexts**
Out[2696]= {"AladjevProcedures`"}
Definition of the global variable**$UserContexts** defining a list of contexts of

the user packages uploaded into the current session completes the previous fragment. At that, the variable determines only contexts of the packages that generate in the current session the variables of two types represented above according to the**UserPackTempVars** procedure. Depending on a state of the current session the execution of the above–mentioned***2*** means can demand certain temporal expense.

Qua of an addition to the above means the **NamesNbPackage** procedure can present a certain interest, whose call**NamesNbPackage[*W*]** returns the list of names in string format of all means which are located in a datafile***W*** of***nb*–**format with a package and that are supplied with***"usages"*.** The next fragment represents source code of the**NamesNbPackage** procedure with an example of its application to***nb*–**file with***AVZ_Package*** package. While the procedure call**NamesNbPackage1[*W*]***(the procedure is an effective enough modification of the previous procedure)* returns the similar list of*names* in string format of all means which are located in a datafile***W*** of***nb***–format with a package; it is supposed that all means are provided with***“usages”;*** in the absence of such means the empty list, i.e. {} is returned.

In[2628] **:= NamesNbPackage[f_ /; IsFile[f] && FileExtension[f]== “nb” && ! SameQ[ContextFromFile[f], $Failed]] := Module[{Res = {}, Tr}, Tr[x_ /; StringQ[x]] :=Module[{c, d, h, g = “"::"”, v = “"="”, p = “"usage"”, a = OpenRead[x], s = ” RowBox[{”}, Label[c]; d = Read[a, String];**

If[d === EndOfFile, Close[a]; Return[Res], Null]; If[DeleteDuplicates[Map3[StringFreeQ, d, {s, g, p, v}]] == {False}&& SuffPref[d, s, 1], h = Flatten[StringPosition[d, g]]; AppendTo[Res, StringTake[d, {12, h[[1]]–3}]]; Goto[c], Goto[c]]];

Map[ToExpression, Sort[Tr[f]]]]

In[2629] **:= NamesNbPackage[“C:\Mathematica\AVZ_Package.nb”]** Out[2629]= {”AcNb“**,** “ActBFMuserQ“**,** “ActCsProcFunc“**,** “ActRemObj“**,** “AddMxFile“**,** “Adrive“**,** “Adrive1“**,** “Affiliate“**,** “Aobj“**, …..**}

In[2630] **:= NamesNbPackage1[f_ /;IsFile[f]&&FileExtension[f]==“nb”&& ! SameQ[ContextFromFile[f], $Failed]] := Module[{c, d, g = “::”, a = OpenRead[f], p = “usage”, v = “=”, Res = {}, s = ” RowBox[{”}, Label[c]; d = Read[a, String]; If[d === EndOfFile, Close[a];**

Return[Sort[Map[ToExpression, Res]]], If[DeleteDuplicates[Map3[StringFreeQ, d, {s, g, p, v}]] == {False}&& SuffPref[d, s, 1], AppendTo[Res, StringReplace[StringSplit[d, “,”][[1]], s–> ””]]; Goto[c]]; Goto[c]]]

In[2631]= **NamesNbPackage1[“C:\Mathematica\AVZ_Package.nb”]** Out[2631]= {”AcNb“**,** “ActBFMuserQ“**,** “ActCsProcFunc“**,** “ActRemObj“**,**

“ AddMxFile“**,** “Adrive“**,** “Adrive1“**,** “Affiliate“**,** “Aobj“**, …..**} In[2632]**:= Length[%]** Out[2632]= 677

The next **NamesMPackage** procedure represents an analog of two previous procedures**NamesNbPackage** and**NamesNbPackage1,** oriented on a case of the user packages located in datafiles of***m***–format. Successful procedure call**NamesMPackage1[*x*]** returns the list of names in string format of means which are located in a datafile***x*** of***m***–format with a package; it is supposed that all means are provided with***“usages”;*** in the absence of such means the empty list, i.e. {} is returned. The following fragment represents source code of the**NamesMPackage** procedure with an example. This procedure well supplements the procedures**NamesNbPackage** and**NamesNbPackage1.**

In[3342] **:= NamesMPackage[f_ /; IsFile[f] && FileExtension[f] == “m” && ! SameQ[ContextFromFile[f], $Failed]] := Module[{c, d, Res = {}, s = “::usage="”, a = OpenRead[f]}, Label[c]; d = Read[a, String]; If[SuffPref[d, “(*Begin["`”, 1] ||**

d === EndOfFile, Close[a]; Return[Sort[DeleteDuplicates[Res]]], If[SuffPref[d, “(*”, 1] && ! StringFreeQ[d, s], AppendTo[Res, StringTake[d, {3, Flatten[StringPosition[d, s]][[1]]–1}]]; Goto[c], Goto[c]]]]

In[3343]**:= NamesMPackage[“C:\AVZ_Package\AVZ_Package_1.m”]** Out[3343]= {”AcNb“**,** “ActBFMuserQ“**,** “ActCsProcFunc“**,** “ActRemObj“**,**

“ AddMxFile“, “Adrive“, “Adrive1“, “Affiliate“, “Aobj“, ….. , “WhichN“, “XOR1“, “$AobjNobj“, “$CallProc“, “$InBlockMod“, “$Line1“, “$Load$Files$”, “$ProcName“, “$ProcType“, “$TestArgsTypes“, “$TypeProc“, “$UserContexts”}

In[3344]:= **Length[%]**
Out[3344]= 682

The **ContextFromFile** function presented in the above fragment generalizes ***3*** procedures**ContextMfile, ContextMXfile** and**ContextNBfile,** returning the context associated with the packages saved in datafiles of format {***cdf,m, mx, nb***}, and**$Failed** otherwise.

The question of extraction of definitions of functions and procedures from an unuploaded package which is located in a datafile of***m***–format is rather actual. In this regard we will present a procedure which solves this problem for the package, located in a datafile of format {***“cdf”,“nb”***}. The principal organization of a datafile of these formats with a package is represented at the beginning of the next fragment that is used and as one of examples. This package is previously saved in a datafile of format {***“cdf”, “nb”***} by a chain of the commands***“File→Save As”*** of the***GUI****(Graphic User Interface).*

```
BeginPackage[“Grodno`”]
Gs::usage = “Help on Gs.”
Ga::usage = “Help on Ga.”
Vgs::usage = “Help on Vgs.”
GSV::usage = “Help on GSV.”
Begin[“`Private`”]
Sv[x_] := x^2 + 26*x + 18
End[]
Begin[“`Gs`”]
Gs[x_Integer, y_Integer] := x^2 + y^2
End[]
Begin[“`Ga`”]
Ga[x_Integer, y_Integer] := x*y + Gs[x, y]
End[]
Begin[“`Vgs`”]
Vgs[x_Integer, y_Integer] := x*y
End[]
Begin[“`GSV`”]
GSV[x_Integer, y_Integer] := Module[{a = 90, b = 500, c = 2015},

x*y + Gs[x, y]*(a+b+c)] +a*Sin[x]/(b+c)*Cos[y] End[]
EndPackage[]
```

In[2669]:= **ExtrFromNBfile[x_ /; FileExistsQ[x] && MemberQ[{*”cdf“,”nb”*},**

FileExtension[x]], n_/; StringQ[n]] := Module[{a = ToString[InputForm[Get[x]]], b = “`” <> n <> “`”,
c = “RowBox[List[RowBox[List[“, k}, a = StringReplace[a, {“"\[IndentingNewLine]"”–> ””, “"\n"”–> ””}]; If[StringFreeQ[a, b], $Failed, a = StringTake[a, {Flatten[StringPosition[a, b]][[1]] + 1,–1}]; a = StringTake[a,

{Flatten[StringPosition[a, c]][[1]],–1}]; c = StringTake[a, {1, Flatten[StringPosition[a, "RowBox[List["End",”]][[1]]–1}]; For[k = StringLength[c], k >= 1, k—, c = StringTake[c, {1, k}]; If[! SameQ[Quiet[ToExpression[c]], $Failed], Break[]]]; c = Quiet[ToString[InputForm[ToExpression[c]]]]; c = StringReplace[c, {“\(“–> ””, “\)”–> ””}]; If[SuffPref[c, “RowBox[{“, 1] && SuffPref[c, “, Null}]”, 2], StringTake[c, {9,–9}]]; If[SuffPref[c, “RowBox[{“, 1], $Failed, Quiet[Check[ToExpression[c], Return[$Failed]]]; c]]]

In[2670]:= **ExtrFromNBfile[“C:/Mathematica/AVZ_Package.nb”, “StrStr”]**
Out[2670]= **“**StrStr[x_]:= If[StringQ[x], "\"" <> x<> "\"",

ToString[x]] **“**
In[2671]**:= ExtrFromNBfile[“C:/Mathematica/AVZ_Package.cdf”, “StrStr”]**
Out[2671]= **“**StrStr[x_]:= If[StringQ[x], "\"" <> x<> "\"",

ToString[x]]
In[2672]**:= ExtrFromNBfile[“C:\Mathematica\Grodno.nb”, “GSV”]** Out[2672]= **“**GSV[x_Integer, y_Integer]:= Module[{a= 90, b= 500,

c = 2015}, x*y+ Gs[x, y]*(a+ b+ c)]+ a*Sin[x]/(b+ c)*Cos[y]**”** The successful procedure call**ExtrFromNBfile[*x, y*]** returns the definition of an object in the string format with a name***y*** given in string format from an unuploaded datafile***x*** of format {***“cdf”,“nb”***}, at the same time activating this definition in the current session; otherwise, the call returns**$Failed.** Qua of an useful property of this procedure is the circumstance that a datafile***x*** not require of uploading into the current**Mathematica** session.

The next **ExtrFromMfile** procedure is specific complement of the previous **ExtrFromNBfile** procedure, providing extraction of definitions of functions and procedures along with their usages from an unuploaded package that is located in a datafile of***m***–format. The procedure call**ExtrFromMfile[*x,y*]** returns the definition of an object in the string format with a name or list of their names***y*** given in string format from an unuploaded file***x*** of***m***format, at the same time activating these definitions and usages corresponding to them in the current session; otherwise, the call returns empty list, i.e. {}. The following fragment represents source code of the**ExtrFromMfile** procedure along with typical examples of its usage.

In[2608] **:= ExtrFromMfile[x_ /; FileExistsQ[x] && FileExtension[x] == “m”, y_ /; SymbolQ[y] || ListQ[y] && DeleteDuplicates[Map[SymbolQ, y]] == {True}] := Module[{a = ReadString[x], b, c, d, d1, n}, b = StringSplit[a, {“(**)”, “(* ::Input:: *)”}]; b = Map[If[! StringFreeQ[#, {“::usage=”, “BeginPackage["”, “End[]”}], #, Null] &, b]; b = Select[b, ! SameQ[#, Null] &]; c =Map[ToString, Flatten[{y}]]; d = Map[“Begin["`” <># <> “`"]” &, c]; d1 = Map[“(*” <> # <> “::usage=” &, c]; b = Select[b, ! StringFreeQ[#, Join[d, d1]] &]; b = Map[StringTake[#, {3,–5}] &, b];**

c = Map[If[SuffPref[#, d1, 1], StringTake[#, {3,–1}], n = StringReplace[#, GenRules[d, ””]]; n = StringReplace[n, “*)\r\n(*”–> ””];

StringTake[n, {3,–6}]] &, b]; ToExpression[c]; c]]

In[2609] **:= ExtrFromMfile[“C:/Temp/AVZ_Package.m”, {StrStr, HowAct}]**

Out[2609]= {”HowAct::usage="The call HowAct[Q] returns the value True if Q is an object active in the current session, and the False otherwise. In many cases the procedure HowAct is more suitable than standard function ValueQ, including local variables in procedures.“, “StrStr::usage="The call StrStr[x] returns an expression x in string format if x is different from string; otherwise, the double string obtained from an expression x is returned.“,

“ StrStr[x_]:=If[StringQ[x],"\""<>x<>"\"",ToString[x]]“, “HowAct[x_]:=If[Quiet[Check[ToString[Definition[x]],True]]==="Null", False,If[Quiet[ToString[Definition[P]]]==="Attributes["<>ToString[x] <>"]={Temporary}",False,True]]”}

In[2610] **:= ExtrFromMfile[“C:\Temp\AVZ_Package.m”, StrStr]** Out[2610]= {”StrStr::usage="The call StrStr[x] returns an expression x in string format if x is different from string; otherwise, the double string obtained from an expression x is returned.“,
“StrStr[x_]:=If[StringQ[x],"\""<>x<>"\"",ToString[x]]”}

The problem of editing of a package that is located in a ***m***file is interesting enough; the following**RedMfile** procedure solves the given problem whose source code with typical examples of use represents the following fragment.

In[2864]:= **PosListTest[l_List, p_ /; PureFuncQ[p]] := Module[{a ={}, k = 1}, While[k <=Length[l], If[Select[{l[[k]]}, p]!={}, AppendTo[a, k]]; k++]; a]** In[2865]:= **PosListTest[{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 18, 26}, EvenQ[#] &]** Out[2865]= {2, 4, 6, 8, 10, 11, 12}
In[2866]:= **RedMfile[x_ /; FileExistsQ[x] && FileExtension[x] == “m”, p_ /; SymbolQ[p], r_ /; MemberQ[{“add”, “delete”, “replace”}, r]] :=**

Module[{a = ReadList[x, String], d = ToString[p], h, save, b = “(*Begin["`” <> ToString[p] <> “`"]*)”, c = “(*End[]*)”}, If[MemberQ[! ContentOfMfile[x], ToString[p]] && r == “delete” || MemberQ[{“add”, “replace”}, r]&& ! (ProcQ[p]||QFunction[p]), $Failed, save[q_] := Module[{f, k = 1}, f = DirectoryName[x] <> FileBaseName[x] <> “$.m”; While[k <= Length[q], WriteString[f, q[[k]], “\n”]; k++]; Close[f]]; If[! MemberQ[a, “(* ::Package:: *)”], $Failed,

If[r === “delete”, h = Select[a, SuffPref[#, “(*” <> d <> “::usage”, 1] &]; If[h == {}, x, a = Select[a, ! SuffPref[#, “(*” <> d <> “::usage”, 1] &]; d = SubListsMin[a, b, c, “r”]; d = MinusList[a, d]; save[d]], If[r === “add” && Select[a, SuffPref[#, “(*” <> d <> “::usage=”, 1] &] == {} && Head[p::usage] == String && (ProcQ[p] || FunctionQ[p]), h = PosListTest[a, SuffPref[#, {“(*BeginPackage[“, “(*EndPackage[]”}, 1] &]; a = Insert[a, “(*” <> d <> “::usage= ” <> ToString1[p::usage] <> “*)”, h[[1]] + 1]; a = Flatten[Insert[a, {“(*Begin["`” <> d <> “`"]*)”, “(*” <> PureDefinition[p] <> “*)”, “(*End[]*)”}, h[[2]] + 1]]; save[a], If[r === “replace” && Head[p::usage] == String && (ProcQ[p] || FunctionQ[p]), h = PosListTest[a, SuffPref[#, “(*Begin["`” <> d <> “`"]*)”, 1] &]; If[h == {}, $Failed, a[[h[[1]] ;; h[[1]] + 2]] = {“(*Begin["`” <> d <> “`"]*)”, “(*” <> PureDefinition[p] <> “*)”, “(*End[]*)”}; h = PosListTest[a, SuffPref[#, “(*” <> d <> “::usage=”, 1] &];

a[[h[[1]]]] = "(*" <> d <> "::usage= " <> ToString1[p::usage] <> "*)"; save[Flatten[a]]]]], x]]]]

(* ::Package:: *) ***Contents of the initial m–file*** **(* ::Input:: *)**
(*BeginPackage["Grodno`"]*)
(*Gs::usage = "Help on Gs."*)
(*Vgs::usage = "Help on Vgs."*)
(*Begin["`Gs`"]*)
(*Gs[x_Integer, y_Integer] := x^2 + y^2*)
(*End[]*)
(*Begin["`Vgs`"]*)
(*Vgs[x_Integer, y_Integer] := x*y*)
(*End[]*)
(*EndPackage[]*)

In[2867] **:= Avz[x_] := Module[{}, x^2 + 90]; Vgs[x_, y_] := x^2 + y^2** In[2868]**:= Avz::usage = "Help on Avz."; Vgs::usage = "Help on Vgs_1.";** In[2869]**:= RedMfile["C:\Mathematica\Grodno.m", Vgs, "delete"]** Out[2869]= **"**C**:**\Mathematica\Grodno**$**.m**"**
(* ::Package:: *) ***Contents of m–file after the operation "delete"*** **(* ::Input:: *)**
(*BeginPackage["Grodno`"]*)
(*Gs::usage = "Help on Gs."*)
(*Begin["`Gs`"]*)
(*Gs[x_Integer, y_Integer] := x^2 + y^2*)
(*EndPackage[]*)
In[2870]:= **RedMfile["C:\Mathematica\Grodno.m", Avz, "add"]** Out[2870]= **"**C**:**\Mathematica\Grodno**$**.m**"**

(* ::Package:: *) ***Contents of m–file after the operation "add"*** **(* ::Input:: *)**
(*BeginPackage["Grodno`"]*)
(*Avz::usage = "Help on Avz."*)
(*Gs::usage = "Help on Gs."*)
(*Vgs::usage = "Help on Vgs."*)
(*Begin["`Gs`"]*)
(*Gs[x_Integer, y_Integer] := x^2 + y^2*)
(*End[]*)
(*Begin["`Vgs`"]*)
(*Vgs[x_Integer, y_Integer] := x*y*)
(*End[]*)
(*Begin["`Avz`"]*)
(*Avz[x_] := Module[{}, x^2 + 90]*)
(*End[]*)
(*EndPackage[]*)

In[2871] **:= RedMfile["C:\Mathematica\Grodno.m", Vgs, "replace"]** Out[2871]= **"**C**:**\Mathematica\Grodno**$**.m**"**
(* ::Package:: *) ***Contents of m–file after the operation "replace"*** **(* ::Input:: *)**
(*BeginPackage["Grodno`"]*)

```
(*Gs::usage = "Help on Gs."*)
(*Vgs::usage = "Help on Vgs_1."*)
(*Begin["`Gs`"]*)
(*Gs[x_Integer, y_Integer] := x^2 + y^2*)
(*End[]*)
(*Begin["`Vgs`"]*)
(*Vgs[x_, y_] := x^2 + y^2*)
(*End[]*)
(*EndPackage[]*)
In[2872]:= RedMfile["C:\Mathematica\Grodno.m", Gs, "add"] Out[2872]=
"C:\Mathematica\Grodno$.m"
In[2873]:= RedMfile["C:\Mathematica\Grodno.m", GsArtKr, "add"] Out[2873]=
$Failed
```

First of all, the previous fragment is preceded by a rather simple procedure, whose call**PosListTest[*l*,*p*]** returns the list of positions of a list***l*** that satisfy the test defined by a pure function***p*.** Further it is supposed that a datafile***x*** of***m***–format structurally corresponds to the standard file with a package; an example of such datafile of***m***–format is given in the first shaded area of the previous fragment. The procedure call**RedMfile[*x*,*n*,*y*]** returns the full path to a***m***–file, whose**FileBaseName** has view**FileBaseName[*x*] <>"$"** which is a result of application to an initial***m***–file of an operation***y*** concerning its object determined by a name***n*,** namely:

"delete"* –** from ax*** datafile the usage and definition of object with a***n*** name are removed, the initial datafile doesn't change; if such object in the datafile is absent, the full path to the initial datafile is returned;

"add"* –** usage and definition of object with an*** name are added into a***x*** file whereas the initial datafile doesn't change; if such object in the file already exists, the full path to the initial datafile***x*** is returned;

"replace"* –** usage and definition of object with an*** name are replaced in a***x*** file while the initial datafile doesn't change; if such object in a file is absent, **$Failed** is returned.

If an initial datafile ***x*** has structure, different from specified, the procedure call returns**$Failed;** at that, successful performance of the operations***"add"*** and***"replace"*** requires preliminary evaluation in the current session of the construction***n::usage*** along with definition for object***n*** as illustrate example of the previous fragment. At that, if an object***n*** is undefined the procedure call returns**$Failed.** In general the procedure allows a number of interesting extensions and modifications which we leave to the interested reader.

Absolutely other situation if necessary to update an object from a package which is located in a datafile of***mx***–format. In this case the next scheme can be used, namely: on the***first*** step the function call**Get[*x*]** uploads into the current session a datafile***x*** of***mx***–format with a package what provides the availability of all means contained in it. While on the***second*** step the usage and definition of an object*(function or procedure)* which should be subjected to updating along with result of a concrete call of this object are checked. On the following step from the current session by means of the**Clear** function the demanded object is removed and for it a new usage is defined. Then, a new definition for the object whose all parameters, including local variables and formal arguments, will be linked with a package context is calculated, accepting the following format, namely**:**

Context_from_File`Object_Name`Variable_of_New_Definition

Then by means of the function call **DumpSave[*y, "Context'"*]** definitions of all objects of the current session that are supplied with a context***"Context'",*** together with their usages are saved in a new file***y*** of***mx*–**format. At last, the final stage in a new current session tests the correctness of the received datafile***y*** of***mx*–**format with the package**–** of a result of modification of an initial datafile***x*** of***mx*–**format with a package. With rather obvious changes the above algorithm quite successfully works and in case of modification of datafiles of***mx*–**format with a package on the basis of operations of addition and removal. The represented algorithm is a rather simple, however has a shortcoming if necessary to modify a datafile of***mx*–**format with a package by means of quite large source codes of objects**;** for similar case a reception described in [30-33] can be used. Meanwhile, it must be kept in mind, the represented algorithm of modification of***mx*–**files with packages belongs to a case when files of***mx*–**format belong to the same operational platform, as their planned modification.

The following **RedMxFile** procedure provides automation of a modification of datafiles of***mx*–**format which is considered above. The call**RedMxFile[*x, y, r, f*]** returns the full path to a***mx*–**datafile, whose**FileBaseName** has view **FileBaseName[*x*] <>"$"** that is a result of application to an initial***mx*–**file of an operation***r*** concerning its object determined by a name***y*,** namely**: *"delete"* –** from a***x*** datafile the usage and definition of object with a***y*** name are removed, the initial datafile doesn**'**t change**;** if such object in the datafile is absent, the full path to the initial datafile is returned**;**
"add"* –** usage and definition of object with ay*** name are added into a***x*** file whereas the initial datafile doesn**'**t change**;** if such object in the file already exists, the full path to the initial datafile***x*** is returned**;** the fourth argument***f*** defines a***mx*–**file containing a package with the usage and definition of the supplemented object***y;***
"replace"*–** usage and definition of object with ay*** name are replaced in a***x*** file while the initial datafile doesn**'**t change**;** if such object in a file is absent, the full path to the initial datafile***x*** is returned**;** the*fourth* argument***f*** defines a***mx*–**file containing a package with the usage and definition of the added***y*** object. At that, if an object***y*** is undefined the procedure call returns**$Failed.** Thus, return of the*path* to an updated datafile***"x$.mx"*** serves as an indicator of success of the**RedMxFile** procedure call. At that, successful performance of the operations***"add"*** and***"replace"*** requires preliminary evaluation in the current session of a construction***y::usage*** along with definition for an object ***y;*** if an object***y*** is undefined the procedure call returns**$Failed.** Source code of the procedure**RedMxFile** along with some typical examples of its usage the following fragment represents.

In[2632]:= **RedMxFile[x_ /; FileExistsQ[x] && FileExtension[x] == "mx", y_ /; StringQ[y] && SymbolQ[y], r_ /; MemberQ[{"add",**

"delete", "replace" }, r], f___] := Module[{a, c, c1 = ContextFromFile[x], c2, save, t}, If[! (ProcQ[y]||QFunction[y]), $Failed, Get[x]; a = CNames[c1]; save[z_] := Module[{p =DirectoryName[z] <> FileBaseName[z] <>"$.mx"}, ToExpression["DumpSave[" <> ToString1[p] <> "," <> ToString1[c1] <> "]"]; p]; If[r == "delete" && MemberQ[a, y], Unprotect[y]; ClearAll[y]; c = save[x];

RemovePackage[c1]; c, If[r == “replace” && MemberQ[a, y] && {f}!= {}&& FileExistsQ[f] && FileExtension[f] == “mx”,

c2 = ContextFromFile[f]; Get[f]; c = ToString1[Definition[y]]; Map[Clear, Mapp[StringJoin, $ContextPath, y]]; Quiet[ToExpression[c1 <> StringReplace[c, c2–> c1]]]; ToExpression[c1 <> y <> “::usage = ” <> ToString1[Help]]; c = save[x]; Map[RemovePackage, {c1, c2}]; c, If[r == “add” && {f}!= {}&& FileExistsQ[f] &&

FileExtension[f] == “mx” && ! MemberQ[a, y], c2 = ContextFromFile[f]; Get[f]; c = ToString1[Definition[y]]; Quiet[Map[Remove, Mapp[StringJoin, $ContextPath, y]]]; Quiet[ToExpression[c1 <> StringReplace[c, c2–> c1]]]; ToExpression[c1 <> y <> “::usage = ” <> ToString1[Help]]; c = save[x]; Map[RemovePackage, {c1, c2}]; c, x]]]]

In[2633] **:= RedMxFile[“C:\Mathematica\Grodno.mx”, “GSV”, “delete”]** Out[2633]= “C:\Mathematica\Grodno$.mx”
In[2634]:= **Get[“C:\Mathematica\Grodno$.mx”]**
In[2635]:= **CNames[ContextFromFile[“C:\Mathematica\Grodno$.mx”]]** Out[2635]= {”Ga“, “Gs“, “Vgs”}
In[2636]:= **Help = “A new help on GSV.”**
Out[2636]= “A new help on GSV.”
In[2637]:= **RedMxFile[“C:\Mathematica\Grodno.mx”, “GSV”, “replace”,**

“C:\Mathematica\GSV.mx”]
Out[2637]= “C:\Mathematica\Grodno$.mx”
In[2638]:= **Get[“C:\Mathematica\Grodno$.mx”]**
In[2639]:= **?GSV**

A new help on GSV.
In[2640]:= **Definition[“GSV”]**
Out[2640]= GSV[x_Integer, y_, z_Integer]:= Module[{a= 90}, (x*y)*a] In[2641]:= **Help = “Help on GSV1.”**
Out[2641]= “Help on GSV1.”

In[2642] **:= RedMxFile[“C:\Mathematica\Grodno.mx”, “GSV1”, “add”, “C:\Mathematica\GSV1.mx”]**
Out[2642]= “C:\Mathematica\Grodno$.mx”
In[2643]:= **Get[“C:\Mathematica\Grodno$.mx”]**
In[2644]:= **CNames[ContextFromFile[“C:\Mathematica\Grodno$.mx”]]**
Out[2644]= {”Ga“, “Gs“, “GSV“, “GSV1“, “Vgs”}
In[2645]:= **?GSV1**
Help on GSV1.
In[2646]:= **DefFunc[GSV1]**
Out[2646]= GSV1[x_Integer, y_]:= Module[{a= 47}, x*y*a]

So, for providing of the operation ***“add”*** or***“replace”*** a datafile of***mx***-format with a package should be previously created that contains definition of an object used for updating*(replacement,addition)* of a main***mx***-datafile with the package. At the same time it must be kept in mind that both updating and updated***mx***–files have to be created on the same operational platform. At that, qua of the result of a procedure call both packages are removed from the current session. In general, the**RedMxFile** procedure allows a number of extensions which we leave to the interested reader. Meanwhile, it should be noted, this procedure in a number of the relations is based on receptions, artificial for the standard procedural paradigm providing correct procedure calls in the environment of the system dependent on its version.

A quite useful procedure provides converting of a package located in a file of***mx***–format into a file of***m***–format. The call**MxFileToMfile[*x,y*]** returns the path to a datafile***y*** which is the result of converting of a***mx***–file***x*** with a package into a datafile***y*** of***m***–format. At that, the procedure call deletes the above packages***x, y*** from the current session. The next fragment represents source code of the procedure with an example of application, whereas with the examples of the contents of the initial and converted datafiles***x,y*** with the package the interested reader can familiarize in our books [30–33].

In[2672] **:= MxFileToMfile[x_ /; FileExistsQ[x] && FileExtension[x] ==“mx”, y_ /; StringQ[y] && FileExtension[y] == “m”] := Module[{a = ContextFromFile[x], b, c, k = 1}, Get[x]; b = CNames[a]; WriteString[y, “(* ::Package:: *)”, “\n”, “(* ::Input:: *)”, “\n”, “(*BeginPackage["” <> a <> “"]*)”, “\n”]; While[k <= Length[b], c =b[[k]] <> “::usage”; WriteString[y, “(*” <> c <> ” = ” <>**

ToString1[ToExpression[a <> c]], “*)”, “\n”]; k++]; k = 1; While[k <= Length[b], c = b[[k]]; WriteString[y, “(*Begin["`” <> c <> “`"]*)”, “\n”, “(*” <> PureDefinition[a <> c] <> “*)”, “\n”, “(*End[]*)”, “\n”]; k++]; WriteString[y, “(*EndPackage[]*)”, “\n”];

Map[{Clear1[2, a <> # <> “::usage”], Clear1[2, a <> #]}&, b]; $ContextPath = MinusList[$ContextPath, {a}]; Close[y]] In[2673]:= **MxFileToMfile[“C:\Mathematica\Grodno.mx”, “Tallinn.m”]** Out[2673]= **“**Tallinn.m**”**

While the **MfileToMx** procedure provides converting of a package located in a datafile of***m***–format into a datafile of***mx***–format. The procedure call **MfileToMx[*x*]** returns the path to a datafile that is the result of converting of a***m***–file***x*** with a package into a file of***mx***–format, whose name coincides with the name of the initial datafile***x*** with replacement of the extension***“m”*** on***“mx”.*** Moreover, the procedure call deletes a package***x*** from the current session if upto the**MfileToMx** procedure call the datafile wasn’t loaded, and otherwise no. The next fragment represents source code of the**MfileToMx** procedure along with a typical example of its usage.

In[2721] **:= MfileToMx[x_ /; FileExistsQ[x] && FileExtension[x] == “m”] := Module[{a = ContextFromFile[x], b, d, c = ToString1[x <> “x”]}, If[MemberQ[$ContextPath, a], ToExpression[“DumpSave[” <> c <> “,” <> ToString1[a] <> “]”]; x <> “x”, b = ReadList[x, String]; d = Select[Map[StringReplace[#, {“(*”–> ””, “*)”–> ””}] &, b[[3 ;;–1]]], # != ”” &]; Quiet[ToExpression[d]]; ToExpression[“DumpSave[” <> c <> “,” <> ToString1[a] <> “]”]; Map[Clear1[2, a <> #] &, CNames[a]]; $ContextPath = MinusList[$ContextPath, {a}]; x <> “x”]];** In[2722]:= **MfileToMx[“C:\Mathematica\Rans_Ian.m”]**

Out[2722] **= “**C:\Mathematica\Rans_Ian.mx**”**
This procedure represents a certain interest in a number of appendices.

The question of *documenting* of the user package is an important enough its component**;** at that, absence in a package of usage for an object contained in it does such object as inaccessible at uploading the package into the current session. So, description of each object of the user package has to be supplied with the corresponding*usage*. At the same time it must be kept in mind that mechanism of*documenting* of the user libraries in the**Maple** system is much more developed, than similar mechanism of*documenting* of the user package in the**Mathematica** system. Thus, if the mechanism of formation of the user libraries in the**Maple** is simple enough, providing simple documenting of library means and providing access both to means of library, and to their references at the level of the system means, in the**Mathematica** system the similar mechanism is absent. Receiving of the usage concerning a***x*** package tool is possible only by means of calls***?x*** or**Information[*x*]** provided that a package has been uploaded into the current session. Meanwhile, in case the package contains enough many means, for obtaining the usages concerning the demanded means it is necessary to be sure in their existence, first of all. The next**PackageUsages** procedure can be rather useful to these purposes, whose source code along with examples of its usage are represented below.

In[5313] **:=PackageUsages[x_ /;FileExistsQ[x]&&FileExtension[x]==“m”]:= Module[{a = StringSplit[ReadString[x], {“(**)”, “*)\r\n(*”}], b, c, d, f}, b = Select[a, ! StringFreeQ[#, {“::usage=”, “::usage = “}] &]; c = FileNameSplit[x]; d = FileBaseName[c[[–1]]] <> “.txt”; f = FileNameJoin[Join[c[[1 ;;–2]], {d}]]; Map[{WriteString[f, StringReplace[#, “::usage”–> ””]], WriteString[f, “\n\n”]}&, b]; Close[f]]**

In[5314] **:= PackageUsages[“AVZ_Package.m”, “AVZ_Package_Usages.txt”]**
Out[5314]= **“**AVZ_Package_Usages.txt**”**
In[5315]**:= PackageUsages[“C:\users\aladjev/mathematica/Tallinn.m”]** Out[5315]= **“**C**:**\users\aladjev\mathematica\Tallinn.txt” Gs= **“**Help on Gs.**”**
Rans= **“**Help on Rans.**”**
Vgs= **“**Help on Vgs.**”**
The procedure call**PackageUsages[*x*]** returns the path to a datafile in which the extension***“m”*** of a datafile***x*** is replaced on***“txt”;*** the received datafile contains usages of the user package formed standardly in the form of a***nb–*** document*(see above)* with the subsequent its saving in a***m–***file***x*** by means of chain of the commands***“File –> Save As”*** of the***GUI.*** The information on the specific package tool***y*** has the format***y* = *“Help on y”*.** The received***txt–*** file allows to look through easily its contents regarding search of necessary means of the user package.

For testing of contents of a datafile of ***mx–***format with the user package in the context of names of means whose definitions are located in this datafile, the**NamesFromMx** procedure is a rather useful means. The procedure call **NamesFromMx[*x*]** returns the list of names in string format of tools whose definitions are located in***x*** datafile of***mx–***format with the user package. If the given package wasn**’**t loaded into the current session, the procedure call leaves it unloaded. Fragment below represents source code of the procedure **NamesFromMx** along with typical examples of its usage.

In[5190]**:= NamesFromMx[x_ /; FileExistsQ[x] && FileExtension[x] == “mx”] := Module[{a = ContextFromFile[x], b}, If[MemberQ[$ContextPath, a], CNames[a],**

Get[x]; b = CNames[a]; Map[Close1[2, a <> #] &, b]; $ContextPath = MinusList[$ContextPath, {a}]; b]]

In[5191] **:= NamesFromMx["C:\Mathematica\AVZ_Package.mx"]** Out[5191]= {"AcNb", "ActBFMuserQ", "ActCsProcFunc", "ActRemObj", "AddMxFile", "Adrive", "Adrive1", "Affiliate", "Aobj", "Aobj1", "Args", "Args1", "ArgsBFM", "ArgsTypes", "Arity", ……}
In[5192]**:= Length[%]**
Out[5192]= 684
In[5193]**:= NamesFromMx["C:\Temp\Mathematica\Grodno.mx"]**
Out[5193]= {"Ga", "Gs", "GSV", "Vgs"}
In[5194]**:= $ContextPath**
Out[5194]= {"AladjevProcedures`", "PacletManager`", "QuantityUnits`", "WebServices`", "System`", "Global`"}
In[5195]**:= Definition[GSV]**
Out[5195]= Null
While the**NamesFromMx1** procedure unlike the**NamesFromMx** procedure doesn't demand for obtaining the list of*names*, whose definitions are located in a***mx***–file with the user package, real uploading into the current session of this datafile. The procedure call**NamesFromMx1[*x*]** returns the list of*names* of means whose definitions are located in a x datafile of***mx***–format with the user package. The fragment below represents source code of the procedure **NamesFromMx1** along with some typical examples of its usage.

In[3570]**:= NamesFromMx1[x_ /; FileExistsQ[x] && FileExtension[x] ==**

"mx"] := Module[{c, d = {}, p, h = "", k = 1, j, m, n, a = ContextFromFile[x], b = ToString[ReadFullFile[x]]}, b = StringJoin[Map[FromCharacterCode, Select[ToCharacterCode[b], # > 32 && # < 128 &]]]; {n, m}= Map[StringLength, {a, b}]; c = Map[#[[1]] + n &, StringPosition[b, a]][[2 ;;–1]]; While[k <= Length[c], For[j = c[[k]], j <= m, j++, p = StringTake[b, {j, j}]; If[p == "`", AppendTo[d, h]; h = ""; Break[], h = h <> p]]; k++]; Sort[MinusList[Select[d, SymbolQ[#] &], {"Private"}]]]

In[3571] **:= NamesFromMx1["C:\Temp\Mathematica\Kiev.mx"]** Out[3571]= {"Art", "Avz", "GSV"}
In[3572]**:= Length[NamesFromMx1["C:\Temp\AVZ_Package.mx"]]** Out[3572]= 428

At that, the procedure call **NamesFromMx1[*x*]**returns only those names of means whose definitions received by means of the**Definition** contain the context associated with a package contained in a***mx***–file***x*.** Whereas on the other side certain modifications of the**NamesFromMx1** procedure allow to obtain more complete list of names of means whose definitions with*context* are located in a datafile***x*** of***mx***–format with a package. The next fragment presents one of such modifications qua of which the procedure acts, whose call**NamesFromMx2[*x*]** returns the list of names in string format of means, whose definitions are located in a***mx***–file with package. Along with sourse code of the procedure the examples of its usage are presented. Meanwhile, the both procedures demand enough considerable temporary expenses on datafiles of***mx***–format with a package of rather large size.

In[3584]**:= NamesFromMx2[x_ /; FileExistsQ[x] && FileExtension[x] ==**

“mx”] := Module[{a = ToString[ReadFullFile[x]], b}, b = Select[ToCharacterCode[a], # == 255 || (# > 31 && # < 123 && ! MemberQ[Flatten[{Range[37, 47], Range[91, 95]}], #]) &]; b = ReduceList[b, 255, 1, 1]; b = Select[Quiet[SplitList[b, 96]], # != {}&];

b = Quiet[Map[FromCharacterCode, b]]; b = DeleteDuplicates[Select[b, SymbolQ[#] &]]; Sort[Select[b, ! MemberQ[{“Private”, “System”}, #] && StringFreeQ[#, {StringTake[ContextFromFile[x], {1,–2}], “ÿ”}] &]]]

In[3585] **:= NamesFromMx2[“C:\Temp\Mathematica\Kiev.mx”]** Out[3585]= {”Art“, “Avz“, “GSV”}
In[3586]**:= Length[NamesFromMx2[“C:\Temp\AVZ_Package.mx”]]** Out[3586]= 438

For the purpose of reduction of temporary expenses, the above algorithm of the**NamesFromMx2** procedure can be modified, using the following means extending the**Mathematica** system. The**Map11** function considered above, and procedure**SplitList1,** given by the fragment below, act as such means.

In[5173] **:= SplitList1[x_/;ListQ[x], y_/; ListQ[y], z_/; ListQ[z]] := Module[{c, a = Map12[ToString, {x, y, z}], b = ToString[Unique[“$”]]}, c = Map11[StringJoin, a, b]; c = Map[StringJoin, c]; c = SubsString1[c[[1]], {c[[2]], c[[3]]}, StringQ[#] &, 0]; ToExpression[Map11[StringSplit, c, b]]]**

In[5174] **:= SplitList1[{x, y, z, a, b, c, d, p, m, n, p, x, y, z, 42, 47, 67, 90, m, n, p}, {x, y, z}, {m, n, p}]**
Out[5174]= {{a, b, c, d, p}, {42, 47, 67, 90}}
In[5174]**:= SplitList1[{x, y, z, a, b, c, d, p, x, y, z, 42, 47, 67}, {x, y, z}, {m, n, p}]**
Out[5174]= {}

The procedure call **SplitList1[*x, y, z*]** returns the sublists of a list***x*** which are limited by its sublists***y*** and***z*** excepting the limiting sublists***y*** and***z.*** In the absence of such sublists the empty list, i.e. {} is returned. Along with that the given procedure extends the above–mentioned**SplitList** procedure. As it was noted, the**Mathematica** system has a large enough number of the global variables that describe, for example, characteristics of the system, an operating platform, the full paths to its main directories along with a number of other indicators of current state of the system. Thus, the user has a quite real possibility quite effectively to develop own means, including the means that extend the possibilities of the system itself. In reality, on the basis of a number of such global variables and a number of enough developed tools it is possible to develop the original means; at that, the development of their analogs in the**Maple** system often demands the more essential efforts and non–standard approaches. Our experience in the given direction confirms the told. Some quite simple examples were given in [25-27] and, most often, they concerned the means of access. Considerable interest for the advanced programming in the system also the problem of definition of a name of the current document {***mwsfile, nb****file*} represents. In the**Maple** system for this purpose the***mwsname*** procedure whose development demanded a certain non-standard approach was created. Whereas the development of similar means for the**Mathematica** appeared much simpler, what the next rather simple**NbName** procedure illustrates, whose source code with examples of usage are represented by the following fragment.

In[3743]:= **NbName[] := Module[{a, b, c, d, k = 1},**
{a, d}= {ToString[Notebooks[]], {}};

{ b, c}= Map3[StringPosition, a, {"<<", ">>"}]; While[k <= Length[b], AppendTo[d, StringTake[a, {b[[k]][[2]] + 1, c[[k]][[1]]–1}]]; k++]; Select[d, SuffPref[#, ".nb", 2] &]]

In[3744]:= **NbName[]**
Out[3744]= {"Search.nb", "LoadF.nb", "ActiveProcs.nb", "Int.nb", "Ver.nb"}
In[3745]:= **AcNb[] := StringSplit[NotebookFileName[], {"\", "/"}][[–1]]** In[3746]:= **AcNb[]**
Out[3746]= "AVZ_Package.nb"

The procedure call **NbName[]** returns the list of*nb*–documents which have been loaded into the current session; at that, their order in the list is defined by order of their uploading into the current session so, that the first element defines the current*nb*–document. In turn, the call**AcNb[]** of rather simple function returns the name of the current document or a package which has been earlier saved in a datafile of the*nb*–format.

For convenience of uploading of a package into the current session the**Need** procedure generalizing in a certain degree the standard**Needs** function can be used. The source code of the**Need** procedure along with examples of its usage are represented by the following fragment.

In[2672] **:= Need[x__] := Module[{a = Directory[], c, p, d = {x}[[1]], f, b = If[Length[{x}] > 1 && StringQ[{x}[[2]]], {x}[[2]], "Null"]}, If[! ContextQ[d], $Failed,**
If[b == "Null", Quiet[Check[Get[d], $Failed]],

If[b != "Null"&& ! MemberQ[{"m", "mx"}, FileExtension[b]], $Failed, If[MemberQ[$Packages, d], True, CopyFile[b, f = a <> "\" <> StringTake[d, {1,–2}] <> "." <> FileExtension[b]]; Get[f]; DeleteFile[f]; True]]]]]

In[2673] **:= Need["Grodno`", "C:\mathematica\Grodno.mx"]** Out[2673]= True
In[2674]:= **$Packages**
Out[2674]= {"Grodno`", "AladjevProcedures`", "GetFEKernelInit`",

" ResourceLocator`", "PacletManager`", "System`", "Global`"} In[2675]:= **Definition[Vgs]**
Out[2675]= Vgs[x_/; IntegerQ[x], y_/; IntegerQ[y]]:= x*y

The procedure call **Need[*x*]** loads a package that corresponds to a*x* context into the current session provided that the corresponding datafile of format {***"m"***|***"mx"***} is located in one of the directories determined by the system variable**$Path** with return*True;* otherwise, the call returns$Failed. Whereas the procedure call**Need[*x*]** loads a package that corresponds to a*x* context into the current session provided that the corresponding datafile of format {***"m"***|***"mx"***} is located or in one of the directories determined by the system variable**$Path,** or is determined by argument***y*** with return*True;* otherwise, the call returns$Failed. So, having created a*nb*–document with definitions of objects, having supplied them with usages with its subsequent*evaluation* and*preservation* by means of function {**Save**|**DumpSave**} in a file of format {***"m"***|***"mx"***} respectively, we have a possibility in the subsequent sessions to upload it into the current session by means of the**Needs** function or the **Need** procedure with receiving access to the program objects contained in it. Moreover, for the purpose of increase of efficiency of uploading of a package it is recommended to use a file of***mx***-format in which it was earlier saved by means of the call**DumpSave[*x*]** where the argument***x*** determines the*context* associated with the saved package. With questions of uploading of the user packages into the current session along with rather useful recommendations the interested reader can familiarize in [28,30-33]. In particular, it should be noted the undesirability of use of identical contexts for the user packages, leading in some cases to unpredictable results, but not all so negatively. For example, such approach can be used for replenishment of a datafile of***mx***–format with a package by new means as the earlier considered**RedMxFile** procedure as the next**AddMxFile** procedure illustrates.

In[4222]:= **AddMxFile[x_ /; FileExistsQ[x] && FileExtension[x] == "mx", y_ /; FileExistsQ[y] && FileExtension[y] == "mx", z_ /; FileExtension[z] == "mx"] :=**

Module[{a = ContextFromFile[x], b = ContextFromFile[y], c = 90}, If[a != b, $Failed, If[MemberQ[$ContextPath, a], c = 500; Quiet[Get[y]], Quiet[{Get[x], Get[y]}]]; ToExpression["DumpSave[" <> ToString1[z] <> "," <> ToString1[a] <> "]"]; If[c == 90, Map[Clear1[2, a <> #] &, CNames[a]]; Quiet[$ContextPath = MinusList[$ContextPath, {a}]]]]]; z]

In[4223] **:= AddMxFile["Tallinn.mx", "Grodno.mx", "Rans.mx"]** Out[4223]= "Rans.mx"
In[4224]**:= Get["Rans.mx"]**
In[4225]**:= $Packages**
Out[4225]= {"Grodno`", "AladjevProcedures`", "GetFEKernelInit`",

" ResourceLocator`", "PacletManager`", "System`", "Global`"} In[4226]**:= CNames["Grodno`"]**
Out[4226]= {"Avz", "GSV", "GSV1", "Gs", "Gs1", "Vgs", "Vgs1"} In[4227]**:= PureDefinition[Vgs1]**
Out[4227]= "Vgs1[x_ /; IntegerQ[x], y_ /; IntegerQ[y]]:= x*y"

In[4230]**:= SaveInMx[x_ /; FileExtension[x] == "mx", y_ /; SymbolQ[y] ||**

ListQ[y] && DeleteDuplicates[Map[SymbolQ[#] &, y]] =={True}, z_ /; ContextQ[z]] := Module[{b, a = Flatten[Select[Map[PureDefinition[#] &, Flatten[{y}]], ! SameQ[#, $Failed] &]]}, Map[ToExpression[z <> #] &, a]; AppendTo[$ContextPath, z]; DumpSave[x, z];]

In[4227] **:= Agn[x_, y_] := Module[{a = 90}, a*(x + y)]; Agn[x_] := x + 500** In[4228]**:= SaveInMx["Grodno.mx", {Avz, Agn}, "Grodno`"];**
In[4229]**:= $ContextPath = MinusList[$ContextPath, {"Grodno`"}];**

Clear[Avz, Agn]; Get["Grodno.mx"]
In[4230]**:= PureDefinition[Agn]**
Out[4230]= {"Agn[x_, y_]:= Module[{a=90}, a*(x+y)]", "Agn[x_]:= x+500"}

The previous fragment represents source code of the **AddMxFile** procedure that uses the mechanism of contexts [28,30-33], whose call**AddMxFile[*x,y,z*]** returns the path to a datafile***z*** **–** result of supplement of a datafile***x*** by tools of a datafile***y;*** all datafiles have***mx***format while the*first**2*** datafiles contain the packages with the same context. At that, if the first package is uploaded into the current session, then the second package***y*** also remains uploaded**;** otherwise, the first***2*** packages are unloaded from the current session. While the procedure call**SaveInMx[*x, y, z*]** returns nothing, saving in a***mx–***file***x*** with***z*** context the definition of a symbol or list of symbols***y*** which have the context***"Global'".*** The**SaveInMx** procedure to a certain extent supplements earlier represented means of the same plan.

In a number of cases exists a need of testing of a file regarding that whether it contains a package. The given problem is solved by quite simple function, whose call**PackageFileQ[*x*]** returns*True* if argument***x*** defines a datafile of formats {***"cdf",“mx", "m", "nb"***} with a package, otherwise*False* is returned.

In[2542]**:= PackageFileQ[x_] := If[StringQ[x] && FileExistsQ[x] && MemberQ[**{**"cdf", "m", "mx", "nb"**}**, FileExtension[x]],**

If[SameQ[ContextFromFile[x], $Failed], False, True], False] In[2543]**:= Map[PackageFileQ,** {**"gru.mx", "pack.m", "pack.nb", "pack.cdf"**}**]** Out[2543]= {True**,** True**,** True**,** True}
The previous fragment represents source code of the**PackageFileQ** function

along with examples of its usage. The given function turned out as an useful means for a number of means of our package***AVZ_Package*** [48].

At last, for convenience of loading of the user package located in a ***mx–***file***x*** into the current session the**LoadPackage** procedure can be used, whose call **LoadPackage[*x*]** returns*Null***,** i.e. nothing, uploading the package into the current session with activation of all definitions which are contained in it in the mode similar to the mode of***Input–***paragraph, i.e. in an optimal format in the above sense*(without package context)****.*** The fragment below represents source code of the**LoadPackage** procedure with examples of its usage.

In[3422] **:= LoadPackage[x_ /; FileExistsQ[x] && FileExtension[x] == "mx"] := Module[**{**a**}**,**
Quiet[ToExpression["Off[shdw::Symbol]"]; Get[x]; a = ToExpression["Packages[][[1]]"]; ToExpression["LoadMyPackage[" <> "\"" <>

x <> "\"" <> "," <> "\"" <> a <> "\"" <> "]"]; ToExpression["On[shdw::Symbol]"]]]

In[3423] **:= LoadPackage["C:\Temp\Mathematica\AVZ_Package.mx"]** In[3424]**:= Definition[StrStr]**
Out[3424]= StrStr[x_]**:=** If[StringQ[x]**, "\""** <> x<> **"\"",** ToString[x]]

Meanwhile it must be kept in mind, in case of uploading in a described way into the current session of other user package the availability in the current session of***AVZ_Package*** package or the activated**LoadPackage** procedure is required. The given means is very convenient at processing of definitions of the package tools in the above optimized format, i.e. without a context.

Tools, presented in this chapter along with other tools of our ***AVZ_Package*** [48] allow to solve a number of important problems of processing of the user packages which are located in datafiles of formats {***"cdf", "mx", "m", "nb"***}. The***AVZ_Package*** package represents toolbox oriented on the wide enough circle of appendices including the system ones. The package represents also quite certain interest from standpoint of useful approaches and receptions used at programming a number of the means entering it, including tools for non–standard processing of the user packages.

8.4. The organization of the user software in the *Mathematica*system

The **Mathematica** no possess comfortable enough tools of the organization of the user libraries as in the case of the**Maple,** creating certain difficulties at the organization of the user software developed in its environment. For saving of definitions of objects and results of calculations the**Mathematica** uses datafiles of various organization. At that, datafiles of text format which not only are easily loaded into the current session, in general are most often used, but also are convenient enough for processing by other known means, for example, word processors. Moreover, the text format provides a simple portability on other computing platforms. One of the main prerequisites of saving in datafiles is possibility of use of definitions and their usages of the ***Mathematica***objects in the subsequent sessions of the system. At that, with questions of standard saving of objects*(modules*,*functions*,*usages*,*etc.)* the interested reader can familiarize in details in [28-33,51-53,60,62,64,67], some of them were considered in the present book in the context of organization of packages whereas here we represent simple means of organization of the user libraries in the**Mathematica** system.

Meanwhile, here it is expedient to make a number of very essential remarks on usage of the above system means. First, the mechanism of processing of erroneous and especial situations represents a rather powerful instrument of programming practically of each quite complex algorithm. However, in the**Mathematica** system such mechanism is characterized by a number of essential shortcomings, for example, successfully using in the***Input***–mode the mechanism of output of messages about erroneous situations {**Off, On**}, in the*body* of procedures such mechanism generally speaking doesn't work as illustrates the following rather simple fragment, namely:

```
In[2602] := Import["D:\Math_myLib\ArtKr_2015.m"]
Import::nffil: File not found during Import. >>
Out[2602]= $Failed
In[2603]:= Off[Import::nffil]
In[2604]:= Import["D:\Math_myLib\ArtKr_2015.m"]
Out[2604]= $Failed

In[2605]:= On[Import::nffil]
In[2606]:= F[x_] := Module[{a}, Off[Import::nffil]; a := Import[x];
On[Import::nffil]; a]
In[2607]:= F["D:\Math_myLib\ArtKr_2015.m"]
Import::nffil: File not found during Import. >>
Out[2607]= $Failed
```

So, at creation of complex enough procedures in which is required to solve questions of blocking of output of a number of erroneous messages, means of the**Mathematica** system are presented to us as insufficiently developed means. The interested reader can familiarize with other peculiarities of the specified system means in [28-33]. Now we will present certain approaches concerning the organization of the simple user libraries in the**Mathematica** system. Some of them can be useful in practical work with the**Mathematica.**

In view of the scheme of the organization of library considered in [22,25 –27] concerning the**Maple** with organization different from the main library, we will present realization of similar user library for a case of the**Mathematica** system. On the*first* step in file system of the computer a directory, let us say, ***"C:\Math_myLib"*** is created which will contain***txt***–files with definitions of the user procedures/functions along with their usages. In principle, it is possible to place any number of definitions into such***txt***–files, however in this case it is previously necessary to call a procedure whose*name* coincides with name of the***txt***–file, whereupon in the current session all procedures/ functions whose definitions are located in the datafile along with usages are available. That is really convenient in the case when in a single datafile are located the main procedure and all means accompanying it, excluding the standard system means.

On the *second* step the procedures/functions together with their usages are created and debugged with their subsequent saving in the required datafile of a library subdirectory, for example**:**

In[2601]**:= NF[x_] := Sin[x]*Cos[x]; ArtKr[x_, y_] := Sqrt[Sin[x] + 90*NF[y]]**
In[2602]**:= NF::usage = "Help on NF."; Rans::usage = "Help on Rans.";**

Rans[x_]:= Module[{}**, x^2]; ArtKr::usage = "Help on function ArtKr.";** In[2603]**:= CreateDirectory["C:\Math_myLib"];**
In[2604]**:= Save["C:\Math_myLib\Userlib.txt",** {**NF, ArtKr, "NF::usage", "ArtKr::usage", "Rans::usage", Rans**}**]**

In[2605] **:= Clear[NF, ArtKr, Rans]; ArtKr::usage = ""; NF::usage = "";** In[2606]**:= ? ArtKr**
In[2607]**:= Definition[ArtKr]**
Out[2607]= Null
In[2608]**:= Get["C:\Math_myLib\NF.txt"];**
In[2609]**:= Definition[ArtKr]**
Out[2609]= ArtKr[x**_,** y_]**:=** Sqrt[Sin[x]+ 47 NF[y]]
In[2610]**:= ?ArtKr**

Help on function ArtKr.

To save the definitions and usages in datafiles of ***txt***–format perhaps in two ways, namely**:** ***(1)*** by the function call**Save,** saving the previously evaluated definitions and usages in a datafile given by its first argument as illustrates the previous fragment**;** at that, saving is made in the***append***-mode**,** or***(2)*** by creating***txt***–files with names of objects and their usages whose contents are formed by means of a simple word processor, for example,***Notepad.*** At that, by means of the**Save** function we have possibility to create libraries of the user means, located in an arbitrary directory of file system of the computer.

The next fragment presents the **CallSave** procedure whose call**CallSave[*x, y, z*]** returns the result of the call***y*[*z*]** of a procedure/function***y*** on a list***z*** of factual arguments passed the*y* provided that object definition*y* with usage are located in a*txt*–file*x* that has been earlier created by the**Save** function. If an object with the given name*y* is absent in a datafile*x***,**the procedure call returns**$Failed.** If a datafile*x* contains definitions of several procedures or functions of the same name*y***,** the procedure call is executed relative to their definition whose*formal* arguments correspond to a list**z** of*actual* arguments. If*y* defines the list, the call returns the names list of all means contained in*x***.** The following fragment represents source code of the procedure**CallSave** along with examples of its usage relative to the concrete datafile created by means of the standard**Save** function.

In[3582] **:= NF[x_] := Sin[x]*Cos[x]; ArtKr[x_, y_] := Sqrt[Sin[x] + 90*NF[y]] NF::usage = "Help on NF."; Rans::usage = "Help on Rans."; Rans[x_] := Module[{}, x^2]; Rans[x_, y_] := Module[{}, x + y]; ArtKr::usage = "Help on ArtKr.";**

In[3583] **:= Save["C:\Math_myLib\Userlib.txt", {NF, ArtKr, "NF::usage", "ArtKr::usage", "Rans::usage", Rans}]**
In[3584]**:= Clear[ArtKr, NF, Rans]; NF::usage = ""; ArtKr::usage = ""; Rans::usage = "";**

In[3585]**:= CallSave[x_ /; FileExistsQ[x], y_ /; SymbolQ[y] || ListQ[y], z_ /; ListQ[z]] := Module[{b, c, d, nf, u, p, t, v, n, a = StringReplace[StringTake[ToString[InputForm[ReadString[x]]],**

{ 2,–2}], "\r\n\r\n"–> "\r\n \r\n"], s = Map[ToString, Flatten[{y}]]}, b =StringSplit[a, "\r\n \r\n"]; n = Select[b, StringFreeQ[#, " /: "]&]; nf[g_] := StringTake[g, {1, Flatten[StringPosition[g, "[", 1]][[1]]–1}]; c = Select[b, SuffPref[#, p = Flatten[{Map4[StringJoin, s, "["], Map4[StringJoin, s, " /: "]}], 1] &]; {d, u, t, v}= {{}, {}, Map[# <> " /: " &, s], {}}; Map[If[SuffPref[#, t, 1], AppendTo[u,

DelSuffPref[StringReplace[StringTrim[#, t], "\" –> ""], "rn", 2]], AppendTo[d, #]] &, c]; Map[ToExpression, {d, u}]; If[d == {}, $Failed, If[Length[d] == 1, Symbol[nf[d[[1]]]][Sequences[z]], If[Length[DeleteDuplicates[Map[nf[#] &, d]]] == 1, Map[Symbol[nf[#]][Sequences[z]] &, d][[1]], Map[nf[#] &, n]]]]]

In[3586] **:= CallSave["C:\Math_myLib\Userlib.txt", {NF, ArtKr, Rans}, {90, 500}]**
Out[3586]= {"NF", "ArtKr", "Rans", "Rans"}
In[3587]**:= CallSave["C:\Math_myLib\Userlib.txt", ArtKr, {90, 500}]**
Out[3587]= Sqrt[Sin[90]+ 90 Cos[500]Sin[500]]
In[3588]**:= CallSave["C:\Math_myLib\Userlib.txt", NF, {90, 500}]**
Out[3588]= NF[90, 500]
In[3589]**:= CallSave["C:\Math_myLib\Userlib.txt", ArtKr, {500}]**
Out[3589]= ArtKr[500]
In[3590]**:= CallSave["C:\Math_myLib\Userlib.txt", NF, {500}]**
Out[3590]= Cos[500] Sin[500]

In[3591] **:= CallSave["C:\Math_myLib\Userlib.txt", Rans, {500}]** Out[3591]= 250 000
In[3592]**:= CallSave["C:\Math_myLib\Userlib.txt", Rans, {90, 500}]** Out[3592]= 590
In[3593]**:= ?ArtKr**

Help on ArtKr.
In[3594]**:= CallSave["C:\Math_myLib\Userlib.txt", Art, {90, 500}]** Out[3594]= $Failed

In[3600]:= **Save2[x_ /; StringQ[x], y_ /; SymbolQ[y] || ListQ[y]] := If[FileExistsQ[x], Save[x, "\r\n \r\n"]; Save[x, y], Save[x, y]]**

In[3601] **:= Avz[x_, y_, z_] := Module[{a = 500, b = 90}, a*b*x*y*z]; Avz::usage = "Help on Avz." ;**
In[3602]**:= Save2["C:/Math_myLib\Userlib.txt", {Avz, "Avz::usage"}]**
In[3603]**:= Clear[Avz]; Avz::usage = "";**
In[3604]**:= ?Avz**
Help on Avz.
In[3605]**:= CallSave["C:\Math_myLib\Userlib.txt", Avz, {73, 90, 500}]**
Out[3605]= 147 825000 000

Meantime, the **CallSave** procedure provides the call of a necessary function or procedure which is located in a***txt***–file created by means of the standard **Save** function**;** at that, such user library rather reminds an archive because doesn**'**t allow the updating. Whereas for extension of such libraries by new tools it is necessary to use the simple**Save2** function, whose call**Save2 [*x, y*]** append to a datafile***x*** the definitions of the means given by a name or their list***y*** in the format convenient for the subsequent processing by a number of means, in particular, by the**CallSave** procedure. In the previous fragment a source code of the**Save2** function with examples of its use are represented. Thus, the similar organization of the user library provides a simple mode of its maintaining whereas the**CallSave** procedure allows extensions on rather broad circle of functions of operating with the user library. In particular, the principle of modification of text datafiles with definitions and usages of the procedures/functions not only is very simple, but allows to keep history of modifications of definitions of library means also that in a number of cases is rather actual. In our opinion, the represented approach quite can be used for the organization of simple and effective user libraries of traditional type.

The mentioned simple approach to the organization of the user means in the **Mathematica** system is only one of possible methods, giving opportunity of creation of own libraries of procedures/functions with access to them at the level of the system means. The interested reader can familiarize with these questions more in details, for example, in our books [30**–**33].

Qua of other rather useful approach we will present the**CALLmx** procedure whose call provides saving in library directory of definitions of objects and their usages in the form of***mx***–datafiles with possibility of their subsequent loading into the current session. The fragment below represents source code of the**CALLmx** procedure along with some typical examples of its usage.

In[4650]**:= NF[x_] := Sin[x]*Cos[x] + x^3**
In[4651]**:= ArtKr[x_, y_] := Sqrt[42*Sin[x] + 47*Cos[y]] + x*y**

In[4652] **:= CALLmx[y_, z_ /; MemberQ[{1, 2}, z], d___] := Module[{c = {}, h, k = 1, s, a =If[{d}=={}, Directory[], If[StringQ[d] && DirectoryQ[d], d, Directory[]]], b = Map[ToString, If[ListQ[y], y, {y}]]}, If[z ==1, While[k <=Length[b], s = b[[k]]; h = a <>"\" <> s <> ".mx"; If[! MemberQ[{"Null", $Failed}, Definition4[s]], ToExpression["DumpSave[" <>ToString1[h] <> "," <> ToString[s] <> "]"]; AppendTo[c, s]]; k++]; Prepend[c, a], While[k <= Length[b], s = b[[k]]; h = a <> "\"**

<> s <> ".mx"; If[FileExistsQ[h], Get[h]; AppendTo[c, s]]; k++]; c]]

In[4653] **:= NF::usage = "Help on NF"; ArtKr::usage = "Help on ArtKr";** In[4654]**:=**
CALLmx[{**NF, ArtKr, "NF::usage", "ArtKr::usage"**}**, 1]** Out[4654]=
{**"**C**:**\Users\Aladjev\Documents**", "**NF**", "**ArtKr**"**} In[4655]**:= Clear[NF, ArtKr,]**
In[4656]**:= CALLmx[**{**NF, ArtKr**}**, 2]**
Out[4656]= {**"**NF**", "**ArtKr**"**}
In[4657]**:= AGN = Sqrt[NF[42.47]^2 + ArtKr[19.89, 19.96]^4]**
Out[4657]= 180 710**.**0
In[4658]**:= ?ArtKr**

Help on ArtKr.
The procedure call**CALLmx[*y, 1, d*]** returns the list whose the first element defines library directory while the others**–** names of objects from argument ***y****(a separate name or their list)* whose definitions are evaluated in the current session**;** in the presence of the evaluated usages for objects they are saved in a datafile too**;** the optional argument***d*** determines a directory in which the evaluated definitions of objects and their usages***y*** in the form of***mx***–files with the names***"Name.mx"*** where***Name*** is the names of the objects defined by argument***y*** will be located***;*** in case of absence of argument***d*** as a library directory a directory determined by the call**Directory[]** is choosen. Whereas the call**CALLmx[*y,2, d*]** provides loading into the current session of objects whose names are defined by an argument***y*** from a library directory defined by the third argument***d;*** in its absence**Directory[]** directory is supposed. At that, it should be noted that in one datafile is most expedient to place only the main procedure and functions associated with it, excepting references on the standard functions. It allows to form procedural files enough simply.

It is possible to present the **UserLib** procedure which supports a number of useful functions as one more rather simple example of maintaining the user libraries. The procedure call**UserLib[*W, f*]** provides a number of important functions on maintaining of a simple user library located in a datafile***W*** of ***txt***–format. Qua of the second actual argument of***f*** the two-element list acts for which admissible pairs of values of elements can be, namely**:**

{ ***"names", "list"***}*–return of the list of objects names**,**whose definitions are located in a library datafile**;**in case of the empty datafile the call is returned unevaluated**;***
{***"print","all"***}*–output to the screen of full contents of a library datafile**W;**in the case of the empty datafile the procedure call is returned unevaluated**;*** {***"print", "Name"***}*–output to the screen of definition of an object with the name **Name**whose definition is in a library datafile**W;**in case of the empty datafile the call is returned unevaluated**;**in the absence in a library datafile**W**of the demanded means the procedure call returns Null**,**i.e. nothing**,**in such case the procedure call prints the message of the following kind**"Name**is absent in Library**W";*** {***"add", "Name"***}*–saving in a library datafile**W**in the append-mode of an object with a name**Name;**the definition of a saved means has to be previously evaluated in the current session in the**Input**–mode**;***
{***"load", "all"***}*–uploading into the current session of all means whose definitions are in a library file**W;**in case of the empty datafile the call is returned unevaluated**;*** {***"load", "N"***}*–uploading into the current session of an object with name**N**whose definition is in a*

*library file**W;**in the case of the empty datafile the procedure call is returned unevaluated**;**in the absence in a library datafile**W**of a demanded tool the procedure call returns Null**,**i.e. nothing**,**in such case the procedure call prints the message of the following kind**“N**is absent in Library**W”.***

In other cases the **UserLib** procedure call is returned unevaluated. There is a good opportunity to extend the procedure with a number of useful enough functions such as**:** deletion from a library of definitions with usages of the specified means or their obsolete versions, etc. With the given procedure is possible to familiarize more in details, for example, in [30–33,48].

The *list* structure of the**Mathematica** system allows to rather easily simulate the operating with structures of other systems of computer mathematics, for example, the**Maple** system. So, in the**Maple** system the tabular structure as one of the most important structures is used which is rather widely used both for the organization of data structures, and for the organization of the libraries of software. The similar tabular organization is widely used for the organization of package modules of the**Maple** along with a number of tools of our***UserLib*** library [47]. For simulation of the main operations with the *tabular* organization similar to the**Maple** system**,**in the**Mathematica** system the**Table1** procedure can be used. The procedure call**Table1[*L,x*]** considers a list***L*** of the***ListList*** type, whose***2***–element {***x, y***} sublists correspond to an {***index, entry***} of the**Maple** tables respectively as the table. As the second***x*** argument can be***(1)*** a list {***a, b***},***(2)*** a word {***“index”*|*“entry”***} along with an expression of other type***(3).*** The procedure call**Table1[*L, x*]** returns the list of***ListList***–type received from an initial list***L*** as follows.

In the case ***(1)*** in the presence in***L*** of a sublist with the first element***a*** it is replaced onto a list {***a, b***}, otherwise it supplements***L;*** if the argument***x*** has view {***a, Null***}, in the presence in***L*** of a sublist with the first element***a*** the sublist is removed. For the case***(2)*** the list {***indices*|*entries***} accordingly of a list***L*** is returned, whereas in the case***(3)*** the procedure call returns an entry for a***x***index if such in this table really exists. On other tuples of the actual arguments the procedure call **Table1[*x, y*]** returns **$Failed.** The following fragment represents source code of the**Table1** procedure together with the most typical examples of its usage. The represented examples of the**Table1** procedure usage very visually illustrate its functionality.

In[4412] **:= Table1[L_/; ListListQ[L], x_] := Module[{a = {}, c = L, d = {}, k = 1, b = Length[L]}, If[ListListQ[L] && Length[L[[1]]] == 2, For[k, k <= b, k++, AppendTo[a, L[[k]][[1]]]; AppendTo[d, L[[k]][[2]]]]; {a, d}= Map[DeleteDuplicates, {a, d}]; If[x === “index”, a,**

If[x === “entry”, d,
If[ListQ[x] && Length[x] == 2,
If[! MemberQ[a, x[[1]]], AppendTo[c, x],

Select[Map[If[#1[[1]] ===x[[1]] && ! SameQ[x[[2]], Null], x, If[#[[1]]===x[[1]] && x[[2]] === Null, Null, #]] &, L], ! SameQ[#, Null]&]], Quiet[Check[Select[Map[If[# [[1]] === x, #[[2]]]&, L], ! SameQ[#, Null] &][[1]], $Failed]]]]], $Failed]]

In[4413] **:= Tab1 := {{a, a73}, {b, b42}, {c, c47}, {Kr, d18}, {Art, h26}}** In[4414]**:= Table1[Tab1, “entry”]**

```
Out[4414]= {a73, b42, c47, d18, h26}
In[4415]:= Table1[Tab1, “index”]
Out[4415]= {a, b, c, Kr, Art}
In[4416]:= Table1[Tab1, {ArtKr, 2015}]
Out[4416]= {{a, a73}, {b, b42}, {c, c47}, {Kr, d18}, {Art, h26}, {ArtKr, 2015}}
In[4417]:= Table1[Tab1, {Kr, 2015}]
Out[4417]= {{a, a73}, {b, b42}, {c, c47}, {Kr, 2015}, {Art, h26}}
In[4418]:= Table1[Tab1, Art]
Out[4418]= h26
In[4419]:= Table1[Vsv, ArtKr]
Out[4419]= $Failed
In[4420]:= Table1[Tab1, {Vsv, Agn}]
Out[4420]= {{a, a73}, {b, b42}, {c, c47}, {Kr, d18}, {Art, h26}, {Vsv, Agn}}
```

On the basis of the tabular organization supported by the **Table1** procedure it is rather simply possible to determine the user libraries. Qua of one of such approaches we will present an example of**LibBase** library whose structural organization has format of the***ListList*** list and whose elements have length two. The principled kind of such library is given below, namely**:**

LibBase := {{***Help***, {***“O1::usage = "Help on O1", …,***
“On::usage = "Help on On"}},
{***O1,PureDefinition***[***O1***]}, {***O2,PureDefinition***[***O2***]},***…***, {***On,PureDefinition***[***On***]}}

The first element of the two **–**element first sublist of the**LibBase** list is***Help*** whereas the second represents the usages list in string format for all objects, whose definitions are in the**LibBase** library**;** at that, their actual presence in the library isn’t required. Other elements of the**LibBase** library**—*element sublists of format {Oj, PureDefinition[Oj]}, whereOj – ajobject name, and PureDefinition[Oj]– its definition, presented in string optimal format. The following fragment represents theTabLib procedure supporting work with the aboveLibBase library along with concrete examples that rather visually clarify the essence of such maintenance.***

```
In[5248]:= LibBase := {{Help, {“NF::usage = "Help on function NF."”,
“ArtKr::usage = "Help on function ArtKr."”}},

{ NF, “NF[x_, y_] := x + y”}, {ArtKr, “ArtKr[x_, y_] := Sqrt[26*x + 18*y]”}}
In[5249]:= DumpSave[“LibBase.mx”, LibBase]
Out[5249]= {{Help, {”NF::usage= "Help on function NF."”,

“ ArtKr::usage= "Help on function ArtKr."”}}, {NF, “NF[x_, y_]:= x+ y”},
{ArtKr, “ArtKr[x_, y_]:= Sqrt[26*x+ 18*y]”}}

In[5250] := TabLib[Lib_ /; FileExistsQ[Lib] && FileExtension[Lib] == “mx”, x_,
y___] := Module[{a = Get[Lib], b, c}, If[MemberQ[{“index”, “entry”}, x],
Table1[LibBase, x], Map[ToExpression, LibBase[[1]][[2]]]; If[ListQ[x] && Length[x]
== 2, c = If[SameQ[x[[2]], Null], x, {x[[1]], PureDefinition[x[[1]]]}]; b =
Table1[LibBase, c]; If[! SameQ[b, $Failed], LibBase = b;
ToExpression[“DumpSave[” <> ToString1[Lib] <> “,” <>

“LibBase]”]], If[StringQ[x] && ! StringFreeQ[x, “::usage = “], c =
```

Quiet[LibBase[[1]][[2]] = AppendTo[LibBase[[1]][[2]], x]; LibBase = ReplacePart[LibBase, {1, 2}–> c]; ToExpression["DumpSave[" <> ToString1[Lib] <> "," <> "LibBase]"], If[Table1[LibBase, x] === $Failed, $Failed, b = Table1[LibBase, x]; If[! SameQ[b, $Failed], ToExpression[b]; x[y]], $Failed]]]]]

In[5251] **:= Clear[LibBase]; TabLib["LibBase.mx", "index"]**
Out[5251]= {Help, NF, ArtKr}
In[5252]:= **TabLib["LibBase.mx", "entry"]**
Out[5252]= {{"NF::usage= "Help on function NF."",

" ArtKr::usage= "Help on function ArtKr.""},
"NF[x_, y_]:= x+ y", "ArtKr[x_, y_]:= Sqrt[26*x+ 18*y]"} In[5253]:= **NF[x_] := Sin[x]*Cos[x] + x^3**
In[5254]:= **ArtKr[x_, y_] := 42*Sin[x] + 47*Cos[y] + x*y**
In[5255]:= **TabLib["LibBase.mx", {ArtKr, PureDefinition[ArtKr]}]** Out[5255]= {{{Help, {"NF::usage="Help on function NF."", "ArtKr::usage="Help on function ArtKr.""}}, {ArtKr, "ArtKr[x_, y_]:= 42*Sin[x]+ 47*Cos[y]+ x*y"}}} In[5256]:= **TabLib["LibBase.mx", {NF, PureDefinition[NF]}]**
Out[5256]= {{{Help, {"NF::usage="Help on function NF."", "ArtKr::usage="Help on function ArtKr.""}}, {NF, "NF[x_]:= Sin[x]*Cos[x]+ x^3"},
{ArtKr, "ArtKr[x_, y_]:= 42*Sin[x]+47*Cos[y]+x*y"}}} In[5257]:= **TabLib["LibBase.mx", "index"]**
Out[5257]= {Help, ArtKr, NF}
In[5258]:= **Clear[ArtKr, LibBase, NF]**
In[5259]:= **TabLib["LibBase.mx", ArtKr, 90.42, 590.2015]**
Out[5259]= 53 374.4
In[5260]:= **TabLib["LibBase.mx", NF, 500.2015]**
Out[5260]= 1.25151*10^8
In[5261]:= **TabLib["LibBase.mx", ArtKr]**
Out[5261]= ArtKr[]
In[5262]:= **TabLib["LibBase.mx", {NF, Null}]**
Out[5262]= {{{Help, {"NF::usage= "Help on function NF."", "ArtKr::usage="Help on function ArtKr.""}}, {ArtKr, "ArtKr[x_, y_]:= 42*Sin[x]+47*Cos[y]+x*y"}}}
In[5263]:= **TabLib["LibBase.mx", Avz42]**
Out[5263]= $Failed

In[5264] **:= TabLib["LibBase.mx", "Avz::usage = "Help on object Avz.""]**
Out[5264]= {{{Help, {"NF::usage="Help on function NF."", "ArtKr::usage="Help on function ArtKr."", "Avz::usage= "Help on object Avz.""}}, {ArtKr, "ArtKr[x_, y_]:= 42*Sin[x]+ 47*Cos[y]+x*y"}}}
In[5265]:= **LibBase**
Out[5265]= {{{Help, {"NF::usage="Help on function NF."", "ArtKr::usage="Help on function ArtKr."", "Avz::usage= "Help on function Avz.""}}, {ArtKr, "ArtKr[x_, y_]:= 42*Sin[x]+ 47*Cos[y]+x*y"}}}
In[5266]:= **??NF**
Help on function NF.
NF[x_]:= Sin[x]Cos[x]+ x^3
In[5267]:= **?ArtKr**

Help on function ArtKr**.**

The main operations with the library organized thus are supported by the **TabLib** procedure whose source code with examples of use are represented by the previous fragment. The procedure call**TabLib[*x*,*y*]** depending on the second argumen*t**y*** returns or the current contents of the library which is in a***mx***-file***x*,** or names of the objects that are in the library, or their definitions, namely**:**

TabLib[*x*, "*index*"]– returns the list of objects names whose definitions are in a library***x*,** including the name**Help** of help base of the library**; TabLib[*x*,"*entry*"]–** returns the list of objects definitions that are contained in a library***x*,** including also the help base**Help** of the library**; TabLib[*x*,** {***N*, *Df***}**]–** returns the contents of a library***x*** after its extension by a new definition***Df*** of an object with a name***N*** if***Df*** is different from*Null**;*** at that, obsolete definition of***N*****–**object is updated**;**
TabLib[*x*,{***N*, *Null***}**]–** returns the contents of a library***x*** as a result of*removal* from it of definition of an object with a name***N;*** at that, its usage remains**; TabLib[*x*,*N*, *y*, *z*, …]–** returns the result of call***N*****[*y*, *z*, …]** of an object***N*** from a library***x;*** if the object***N*** is absent in the library,**$Failed** is returned**; TabLib[*x*,*N*]–** if***N* –** usage on an object***N*,** it supplements the help base of a library***x*** with return of the updated contents of the library. In other cases the procedure call returns**$Failed** or is returned unevaluated. Qua of a certain initial library**LibBase** intended for filling its by necessary contents a***ListList*-**list of the above format is used. An initial library**LibBase** should be defined before the first procedure call**TabLib.** Naturally, for real use of the**TabLib** procedure qua of a ready software for the organization of the user libraries it demands an extension of the functionality, meantime, it is presented as an illustrative example of one of possible approaches to the solution of a task of the organization of the user software. We leave this task for the interested**Mathematica** user as a rather useful practice. In principle, the presented library organization provided by the**TabLib** procedure and that is based on the tabular organization which is supported by the**Table1** procedure represents a certain analog of a**Maple–**package of tabular type. The represented library has only a basic set of functions which meanwhile provides its quite satisfactory functioning. Meanwhile, on the basis of the offered approach quite really to create the fast rather small libraries of the user procedures and functions that will be very convenient in operation. At that, the similar quite simple means can serve as good tools for*maintenance* of the libraries of the user procedures/functions that have a text format, and that are simply edited by usual word processors, for example,***Notepad.*** The interested reader can develop own means of the library organization in the **Mathematica** software, using approaches offered by us along with others. However, exists a problem of the organization of convenient help bases for the user libraries. A number of approaches in this direction can be found in. In particular, on the basis of the list structure supported by the system it is rather simply possible to determine help bases for the user libraries. On this basis as one of such approaches an example of the**BaseHelp** procedure has been represented, whose structural organization has the list format [30-33].

Meanwhile, it is possible to create the help bases on the basis of the packages containing usages on means of the user library which are saved in datafiles of***mx*–**format. At that, for complete library it is possible to create only one help***mx*–**file, uploading it as required into the current session by means of the**Get** function with receiving in the subsequent of access to all*usages* that are in the datafile. The next**Usages** procedure can represent a quite certain interest for the organization of a help database for the user libraries. This procedure provides maintaining a help base irrespective of a library that is rather convenient in a number of cases of organization of the user software. The fragment below represents source code of the**Usages** procedure along with the most typical examples of its usage.

In[3600] **:= G::usage = "Help on function G.";**
V::usage = "Help on function V.";
S::usage = "Help on function S.";
Art::usage = "Help on procedure Art.";
Kr::usage = "Help on procedure Kr.";

In[3601] **:= Usages[x_/; StringQ[x], y___] := Module[{a, b, h = ""}, If[! FileExistsQ[x], Put[x]];**
If[{y}== {}&& ! EmptyFileQ[x], While[! SameQ[h, EndOfFile],

Quiet[ToExpression[h = Read[x, Expression]]]]; Close[x];, If[{y}== {}&& EmptyFileQ[x], $Failed,

If[Quiet[Check[ListQ[y], False]] &&{y}!={}&& ListSymbolQ[y], a = DeleteDuplicates[Select[y, Head[#::usage] === String &]];
If[a != {}, PutAppend[Sequences[Map[ToString[#] <> "::usage = " <> "\"" <> #::usage <> "\"" &, a]], x], $Failed], If[! Quiet[Check[ListQ[y], False]], b = DeleteDuplicates[Reverse[ReadList[x, Expression]]]; Put[Sequences[Select[b, ! SuffPref[#1, Map[ToString[#] <> "::usage" &, Flatten[{y}]], 1] &]], x], $Failed]]]]]

In[3602]**:= Usages["C:/MathLib/HelpBase.m", {Art, Kr, G, V, Art, Kr, Vsv}]**

A new session with the Mathematica system

In[2216]**:= Usages["C:\MathLib\HelpBase.m"]**
In[2217]**:= ?G**

Help on function G **.**
In[2218]**:= Information[V]**
Help on function V**.**
In[2219]**:= ?S**
Help on function S**.**
In[2220]**:= ?Art**
Help on procedure Art**.**
For initial filling of a help database in the current session all known usages on means that are planned on inclusion into the user library are evaluated as illustrates the first***Input*–**paragraph of the previous fragment. Then by the procedure call**Usages[*x, y*]** the saving in a***x*–**file of the***ASCII*** format of all usages relating to software tools that are defined by a list***y*** is provided. At that, saving is executed in the***append*–**mode into the end of the***x*–**file**;** if the *specified* datafile***x*** is absent, the empty***x*–**file is created. While the procedure call**Usages[*x, y, z, …*]** where arguments, since the second, represent names {***y,z, …***} of software tools, deletes from the help database the usages on these means. At last, the procedure call**Usages[*x*]** activates all usages containing in help database***x*** in the

current session, doing them*available* irrespectively from existence of the means described by these usages. The successful call of the**Usages** procedure returns*Null,* i.e. nothing**;** otherwise, value**$Failed** is returned, in particular, in the case of a call**Usages[*x*]** at the absent or empty datafile***x.*** The presented approach is represented as a rather convenient. At that, the history of modifications of a datafile***x*** is saved while qua of active usage the last usage supplementing the datafile acts.

For receiving usages on means that are in packages, it is possible to use the **UsagesMNb** procedure, whose source code along with typical examples of usage, are represented by the following fragment.

In[4242]**:= UsagesMNb[x_ /; FileExistsQ[x] && MemberQ[{"m", "nb"}, FileExtension[x]]] := Module[{a, b, c}, If[FileExtension[x] == "m", a = Select[ReadList[x, String], ! StringFreeQ[#, "::usage="] &]; a = Map[StringTake[#, {3,–3}] &, a];**

a = Map[If[SymbolQ[StringTake[#, {1, Flatten[StringPosition[#, "::usage="]][[1]]–1}]], #] &, a]; Select[a, ! SameQ[#, Null] &], c = "$.m"; b = ContextFromFile[x]; ToExpression["Save[" <> StrStr[c] <> ", " <> StrStr[b] <> "]"]; b = Select[Quiet[ReadList["$.m", Expression]], ! MemberQ[{Null, {Temporary}}, #] &]; DeleteFile["$.m"]; b]]

In[4243]**:= UsagesMNb["C:\users/aladjev/mathematica/avz_package.mx"]**
Out[4243]= UsagesMNb["C**:**\users/aladjev/mathematica/avz_package.mx"]

In[4244] **:= UsagesMNb["C:\users/aladjev/mathematica/avz_package.m"]** Out[4244]= {**"**UprocQ**::**usage="The call UprocQ[x] returns False if x is not a procedure**;** otherwise, two-element list of the format {True**,** {"Module"| "Block"|"DynamicModule"}} is returned."**,** **……**}
In[4245]**:= UsagesMNb["C:/users/aladjev/mathematica/avz_package.nb"]** Out[4245]= {**"**The call Names1[] returns the nested 4-element list, whose the first element defines the list of names of the procedures, the second**–** the list of names of functions/modules, the third element**–** the list of names whose definitions have been evaluated in the current session of the system, while the fourth element determines the list of other names associated with the current session.**", ……**}

The procedure call **UsagesMNb[*x*]** returns the usages list on software of the user package which is in a datafile***x*** of format {***"m", "nb"***}**;** these usages are returned in string format. At that, for a datafile***x*** of***m***format the usages list containing a prefix***"Name::usage="*** is returned while for a datafile***x*** of***nb*–** format the usages list without such prefix is returned. Furthermore, if for a package from a datafile***x*** of***m*–**format its uploading into the current session isn**'**t required, then for a package from a datafile***x*** of***nb***format its uploading is required. Unlike the procedures**HelpPrint, HelpBasePac** the**UsagesMNb** procedure provides possibility of both perusal of help databases of the user packages, and their processing.

At last, the call of the simple function **Usages1[*x*]** provides the output of all usages describing the means contained in the user package associated with a context***x.*** The following fragment represents source code of the**Usages1** function along with a typical example of its usage.

In[2699]:= **Usages1[x_ /; ContextQ[x]] := DeleteDuplicates[Map[{Print[#], ToExpression["?" <> #]}&, CNames[x]]][[1]]**

In[2700] **:= Usages1["AladjevProcedures`"]**
"AcNb"
The call AcNb[] returns full name of the current document earlier saved as a nb–file.

== The above example illustrates the format returned by a function call.

8.5. A package for the*Mathematica*system

The computer mathematics has found application in many fields of science such as physics, mathematics, education, computer sciences, engineering, chemistry, computational biology, technology, etc. Computer mathematics systems***(CMS)*** such as**Mathematica** are becoming more and more popular in teaching, research and industry. So, researchers use known**Mathematica** system as an essential enough means for solving problems related to their various investigations. The system is ideal tool for formulating, solving, and exploring various mathematical models. Its symbolic manipulation facilities extend greatly over a range of the problems that can be solved with its help. Educators in universities and colleges have revitalized traditional curricula by introducing problems and exercises that widely use the**Mathematica'***s* interactive mathematics and physics. While students can concentrate on the more fundamental concepts rather than on various plural tedious algebraic manipulations. Finally, engineers and experts in industries use the system **Mathematica** as an efficient tool replacing many traditional resources such as reference books, spreadsheets, calculators, and programming languages. These users easily solve mathematical problems, creating various projects and consolidating their computations into professional report. Meanwhile, our experience with system**Mathematica** of releases***8 ÷ 10*** enabled us not only to estimate its advantages in regard to other similar***CMS,*** above all the **Maple** system, but has also revealed a number of faults and shortcomings which were eliminated by us. In particular,**Mathematica** does not support a number of functions important for procedural programming and datafiles processing. As a result, the***AVZ_Package*** package oriented on the solution of the above problems was created [33,48]. The given package contains more than***680*** means which eliminate restrictions of a number of standard means of the**Mathematica,** and expand its software environment with new means. In this context, the package can serve as a certain additional tool of modular programming, especially useful in the numerous applications where certain nonstandard evaluations have to accompany programming. At that, means presented in the given package have a direct relationship to certain*principal* questions of procedure–functional programming in**Mathematica,** not only for the decision of applied problems, but, first of all, for creation of software extending frequently used facilities of the system and/or eliminating their defects or extending the system with new facilities. The software presented in this package contains a series of rather useful and effective receptions of programming in the**Mathematica** system, and extends its software which allows in the system to programme the problems of various purpose more simply and effectively. The additional means composing the above package embrace the next sections of the**Mathematica** system, namely**:**

*– additional means in interactive mode of the**Mathematica**system*
*–additional means of processing of expressions in the**Mathematica**system*
*–additional means of processing of symbols and strings in the**Mathematica***
*–additional means of processing of sequences and lists in the**Mathematica***
*–additional means extending the standard**Mathematica**functions or its software as a whole (control structures branching and cycle**,**etc.)*
*–definition of procedures in the**Mathematica**software*
*–definition of the user functions and pure functions in the**Mathematica**software*
*–means of testing of procedures and functions in the**Mathematica**software*
*–headings of procedures and functions in the**Mathematica**software*
*–formal arguments of procedures and functions**;***
*–local variables of modules and blocks**;**means of their processing*
*–global variables of modules and blocks**;**means of their processing*
*–attributes**,**options and values by default for arguments of the user blocks**,** functions and modules**;**additional means of their processing*
*–some useful additional means for processing of blocks**,**functions and modules*
*–additional means of the processing of internal**Mathematica**datafiles*
*–additional means of the processing of external**Mathematica**datafiles*
–additional means of the processing of attributes of directories and datafiles
–additional and some special means of processing of datafiles and directories
–additional means of operating with packages and contexts ascribed to them
*–organization of the user software in the**Mathematica**system.*

This package, is mostly for people who want the more deep understanding in the**Mathematica** programming, and particularly those the**Mathematica** users who would like to make a transition from a user to a programmer, or perhaps those who already have some limited experience in**Mathematica** programming but want to improve their possibilities in the system. Expert **Mathematica** programmers will probably find an useful information too. The archive***AVZ_Package.zip*** with the given package that owns the license *FreeWare* can be freely downloaded from the web-site presented in [48]. The package contains***4*** datafiles, namely**:*****AVZ_Package.cdf, AVZ_Package.mx, AVZ_Package.m, AVZ_Package.nb.*** In particular, for perusal of the package it is possible to use or datafile***AVZ_Package_1.cdf*** with the*CDF Player***,** or file ***AVZ_Package_1.m*** with a word processor, for example,*Notepad*. Such approach allows to satisfy the user on various operation platforms*(**Mac OS X, Windows, Linux, Linux ARM**)*. The package contains more than***680*** tools that eliminate restrictions of a number of standard functions of the system**,** and extend its software with new means. In this context, this package can serve as a tool of programming, especially useful in numerous applications, where certain nonstandard evaluations have to accompany programming. At that, the memory size, demanded for the***AVZ_Package*** package in the **Mathematica*****10.1.0.0****(on Windows7Pro**,**ver. 6.1.7601)* yields the next result**:** In[1]**:= MemoryInUse[]**
Out[1]= 28 784 392
In[2]**:= Get["C:\Users\Aladjev\Mathematica\AVZ_Package.mx"]** In[3]**:= MemoryInUse[]**
Out[3]= 40724 272
In[4]**:= N[(%–%%%)/1024^2]**
Out[4]= 11**.**3868

i.e. in the**Mathematica** our***AVZ_Package*** package demands more***11.4*** MB, whereas quantity of software whose definitions are located in this package, at the moment of its uploading into the current session of the**Mathematica** system is available on the basis of the following very simple calculations: In[1]:= **Get["C:\Users\Aladjev\Mathematica\AVZ_Package.mx"]** In[2]:= **Length[CNames["AladjevProcedures`"]]**
Out[2]= 684
At that it must be kept in mind that debugging of means of the package was carried out on the basis of**Mathematica** of version***10,*** and partially on the basis of version***9.*** Therefore in some cases there can be certain slips at their performance that are rather simply eliminated. Unfortunately, regardless of sufficient stability of the built–in***Math***–language, upon transition from the younger version of the**Mathematica** to more senior a certain adjustment can be needed. As a rule, similar adjustment for the used version of the system **Mathematica** isn't very complex.

References

1. ***Aladjev V.Z., Hunt Ü., Shishakov M.L.****Mathematics on Personal Computer.***–** Gomel**:** BELGUT Press**,** 1996**,** 498 p**.,** ISBN 34206140233*(in Russian).*
2.***Aladjev V.Z., Shishakov M.L.****Introduction into Mathematical Package* ***Mathematica 2.2.–*** Moscow**:** Filin Press**,** 1997**,** 363 p**.,***(in Russian).*
3.***Aladjev V.Z., Hunt Ü.J., Shishakov M.L.****Basics of Computer Informatics: Textbook.***–** Tallinn**:** Russian Academy of Noosphere**&** TRG**,** 1997**,** 396 p.
4.***Aladjev V.Z., Hunt Ü.J., Shishakov M.L.****Basics of Computer Informatics: Textbook.***–** Moscow**,** Filin Press**,** 1998**,** 496 p**.,** ISBN 5895680682*(in Russian).*
5.***Aladjev V.Z., Hunt Ü.J., Shishakov M.L.****Basics of Computer Informatics: Textbook,Second edition.***–** Moscow**,** Filin Press**,** 1999**,** 545 p*.(in Russian).*
6.***Aladjev V.Z., Vaganov V.A., Hunt Ü.J., Shishakov M.L.****Introduction into Environment of Mathematical Package****Maple V.–*** Minsk**:** International Academy of Noosphere**,** 1998**,** 452 p**.,** ISBN 1406425698*(in Russian).*
7.***Aladjev V.Z., Vaganov V.A., Hunt Ü.J., Shishakov M.L.****Programming in Environment of Mathematical Package****Maple V.–*** Minsk–Moscow**:** Russian Ecology Academy**,** 1999**,** 470 p**.,** ISBN 4101212982*(in Russian).*
8.***Aladjev V.Z., Bogdevicius M.A.****Solution of Physical,Technical and Mathematical Problems with****Maple V.–*** Tallinn–Vilnius**,** TRG**,** 1999**,** 686 p.
9.***Aladjev V.Z., Vaganov V.A., Hunt Ü.J., Shishakov M.L.****Workstation for Mathematician.***–** Tallinn–Gomel–Moscow**:** Russian Academy of Natural Sciences**,** 1999**,** 608 p**.,** ISBN 3420614023*(in Russian with English summary).*
10.***Aladjev V.Z., Shishakov M.L.****Workstation of Mathematician.***–** Moscow**:** Laboratory of Basic Knowledge**,** 2000**,** 752 p**.,** ISBN 5932080523*(in Russian).*
11.***Aladjev V.Z., Bogdevicius M.A. Maple 6:****Solution of Mathematical, Statistical,Physical and Engineering Problems.***–** Moscow**:** Laboratory of Basic Knowledge**,** 2001**,**850 p**.,** ISBN 593308085X*(in Russian with English summary).*
12.***Aladjev V.Z., Bogdevicius M.A.****Special Questions of Operation in Software Environment of the Mathematical Package****Maple.–*** Vilnius**:** International Academy of Noosphere**&** Vilnius Gediminas Technical Univ.**,** 2001**,** 208 p.

13.*Aladjev V.Z., Bogdevicius M.A.* *Interactive* ***Maple:*** *Solution of Statistical, Mathematical,Engineering and Physical Problems*.– Tallinn: International Academy of Noosphere, 2001–2002, CD with Booklet, ISBN 9985927710.

14. *Aladjev V.Z., Vaganov V.A., Grishin E.P.* *Additional Software Means of Mathematical Package* ***Maple*** *of releases* ***6*** *and* ***7*.**– Tallinn: International Academy of Noosphere, 2002, 314 p.+ CD, ISBN 9985927737 *(in Russian).* **15.*Aladjev V.Z.*** *Effective Operation in Mathematical Package* ***Maple*.**– Moscow: Laboratory of Basic Knowledge, 2002, 334 p., ISBN 593208118X. **16.*Aladjev V.Z., Liopo V.A., Nikitin A.V.*** *Mathematical Package* ***Maple*** *in Physical Modeling*.– Grodno: Grodno State University, 2002, 416 p. **17.*Aladjev V.Z., Vaganov V.A.*** *Computer Algebra System* ***Maple:*** *A New Software Library*.– Tallinn: International Academy of Noosphere, the Baltic Branch, 2002, CD with Booklet, ISBN 9985927753 *(in Russian).* **18.*Aladjev V.Z., Bogdevicius M.A., Prentkovskis O.V.*** *A New Software for Mathematical Package* ***Maple*** *of Releases* ***6, 7*** *and* ***8*.**– Vilnius: Vilnius Gediminas Technical University **&** International Academy of Noosphere, 2002, 404 p., ISBN 9985927745, 9986055652 *(in Russian with extended English summary).* **19.*Aladjev V.Z., Vaganov V.A.*** *Systems of Computer Algebra:A New Software Toolbox for* ***Maple*.**– Tallinn: International Academy of Noosphere, the Baltic Branch, 2003, 270 p., ISBN 9985927761 *(in Russian with English summary).* **20.*Aladjev V.Z., Bogdevicius M., Vaganov V.A.*** *Systems of Computer Algebra: A New Software Toolbox for* ***Maple*.** *Second edition*.– Tallinn: Intern. Academy of Noosphere, 2004, 462 p., ISBN 9985927788 *(in Russian).*
21.*Aladjev V.Z.* *Computer Algebra Systems:A New Software Toolbox for the* ***Maple*.**– CA: Palo Alto: Fultus Corporation, 2004, 575 p., ISBN 1596820004. **22.*Aladjev V.Z.*** *Computer Algebra Systems:A New Software Toolbox for* ***Maple*.** CA: Palo Alto: Fultus Corporation, 2004, Acrobat eBook, ISBN 1596820152. **23.*Aladjev V.Z. et al.*** *Electronic Library of Books and Software for Scientists, Experts,Teachers and Students in Natural and Social Sciences*.– CA: Palo Alto: Fultus Corporation, 2005, CD, ISBN 1596820136 *(in Russian and English).* **24.*Aladjev V.Z., Bogdevicius M.A. Maple:*** *Programming,Physical and Engineering Problems*.– Palo Alto:Fultus Corp., 2006, 404 p., ISBN 1596820802, *eBook*, ISBN 1596820810, ***http://writers.fultus.com/aladjev/index.html*** **25.*Aladjev V.Z.*** *Computer Algebra Systems.* ***Maple:*** *Art of Programming*.– Moscow: BINOM Press, 2006, 792 p., ISBN 5932081899 *(in Russian).* **26.*Aladjev V.Z.*** *Foundations of programming in* ***Maple:*** *Textbook*.– Tallinn: International Academy of Noosphere, 2006, 300 p., *(pdf),* ISBN 998595081X.

Can be freevely from website ***http://www.aladjev-maple.narod.ru*.** **27.*Aladjev V.Z., Boiko V.K., Rovba E.A.*** *Programming and Applications Elaboration in* ***Maple*.**– Grodno: GRSU, Tallinn: International Academy of Noosphere, 2007, 456 p., ISBN 9789854178912, ISBN 9789985950821. **28.*Aladjev V.Z., Vaganov V.A.*** *Modular Programming:* ***Mathematica vs Maple,*** *and vice versa*.– CA: Palo Alto, Fultus Corporation, 2011, 418 p. **29.*Aladjev V.Z., Bezrukavyi A.S., Haritonov V.N., Hodakov V.E.*** *Programming:* ***Maple*** *or* ***Mathematica?*–** Ukraine: Herson, Oldi–Plus Press, 2011, 474 p., ISBN 9789662393460 *(in Russian with English summary).* **30.*Aladjev V.Z., Boiko V.K., Rovba E.A.*** *Programming in the Packages* ***Mathematica*** *and* ***Maple:*** *Comparative Aspect*.– Belarus: Grodno, Grodno State University, 2011, 517 p., ISBN 9789855154816 *(in Russian).* **31.*Aladjev V.Z., Grinn D.S., Vaganov V.A.*** *The extended functional means for the package* ***Mathematica*.**– Ukraine: Kherson:

Oldi–Plus Press, 2012, 404 p., ISBN 9789662393590*(in Russian with extended English summary).* **32.*Aladjev V.Z., Grinn D.S.***Extension of functional environment of the system* ***Mathematica.–*** Ukraine: Kherson: Oldi–Plus Press, 2012, 552 p., ISBN 9789662393729*(in Russian with extended English summary).*
33.*Aladjev V.Z., Grinn D.S., Vaganov V.A.**The selected system problems in* ***Mathematica****software***.–** Ukraine: Kherson: Oldi–Plus Press, 2013, 556 p., ISBN 9789662890129*(in Russian with extended English summary).* **34.*Aladjev V.Z., Bogdevicius M.****Use of package****Maple****for solution of physical and engineering problems//* Int. Conf.*Transbaltica-**99.–*** Vilnius: Technics Press. **35.*Aladjev V.Z., Hunt U.****Workstation for mathematicians//* Conf.*Transbaltica-*

99 .– Vilnius: Technics Press, April 1999.
36.*Aladjev V.Z., Hunt U.**Workstation for mathematicians//*Conf.*«Perfection of Mechanisms of Management»*, Institute of Modern Knowledge, Grodno, 1999. **37.*Aladjev V.Z., Shishakov M.****Programming in package****Maple//2nd*** Int. Conf.

« Computer Algebra in Fundamental and Applied Researches and Education».– Byelorussia: Minsk, 1999.
38.*Aladjev V.Z., Shishakov M.L.**A Workstation for mathematicians//**2nd*** Conf. *«Computer Algebra in Fundamental and Applied Researches and Education».–* Byelorussia: Minsk, 1999.
39.*Aladjev V.Z., Shishakov M.L., Trokhova T.A.**Educational computer laboratory of the engineer//* Proc. ***8th*** Byelorussia Math. Conf., Minsk, 2000. **40.*Aladjev V.Z. et al.****Modelling in software environment of the mathematical package****Maple//*** Int. Conf. on Math. Mod.*MKMM–2000.–* Herson, 2000. **41.*Aladjev V.Z., Shishakov M.L., Trokhova T.A.****A workstation for solution of systems of differential equations//**3rd*** International Conf. *«Differential Equations and Applications».–* Saint–Petersburg, Russia, 2000.
42.*Aladjev V.Z., Shishakov M.L., Trokhova T.A.**Computer laboratory for engineering researches//* Int. Conf.***ACA-2000.–*** Saint–Petersburg, Russia, 2000 **43.*Aladjev V.Z., Bogdevicius M., Hunt U.J.****A Workstation for mathematicians* / Lithuanian Conf.*TRANSPORT–**2000.–*** Vilnius: Technics Press, April 2000. **44.*Aladjev V.Z.****Computer Algebra//* Alpha, №***1.–*** Grodno: GRSU, 2001. **45.*Aladjev V.Z.****Modern computer algebra for modeling of the transport systems //* Intern. Conf.*TRANSBALTICA–**2001.–*** Vilnius: Technics Press, April 2001. **46.*Aladjev V.Z.****Computer Algebra System****Maple:****A New Software Library//* International Conference*«Computer Algebra Systems and Their Applications»*, Saint–Petersburg, Russia, 2003.
47.*Aladjev V.Z.**A Library****UserLib6789****for system****Maple.****–The library can be freely downloaded from website****http://yadi.sk/d/P1FQaYmW619C7.*** **48.*Aladjev V.Z.****A package****AVZ_Package****for system****Mathematica.****–Package can be freely downloaded from website****http://yadi.sk/d/G9HBFqTILiAAC.*** **49.*Aladjev V.Z.****Modular programming:* ***Maple****or****Mathematica*** *–A subjective standpoint/Intern. school«Mathematical and computer modeling of fundamental objects and phenomena in systems of computer mathematics»*, ed.***Y. G. Ignat'ev.–*** Kazan: Kazan Univ. Press, 2014, pp. 18–32.
50.*Nelson H.F. Beebe.**A Bibliography of Publications about the****Maple****Symbolic Algebra Language.–* Salt Lake City: Univ. of Utah, Dept. of Mathem., 2010. **51.*Arantes R.D.****A Computational Reference Guide on Experimental Mathematics,Algorithmic Number Theory and Symbolic Computing.–* Rio de Janeiro: Federal University, Caixa Postal

11502, 220022–970, 2004. **52.*Mangano S.****Mathematica Cookbook*.– CA: Sebastopol: O'Reilly Media, Inc., 2010, ISBN–13: 9780596520991, ISBN–10: 0596520999, 828 p.
53.*Wellin P.et al.**An Introduction to Programming with**Mathematica**,3rd ed*.– Cambridge University Press, 2005, 550 p., ISBN 0521846781. **54.*Sisson P.****College Algebra,2nd ed.*– Hawkes Learning Systems, 2008.

55. *Gregor J.,Tier J.**Discovering Mathematics:A Problem–Solving Approach to Mathematical Analysis with**Mathematica**and**Maple**.*– Springer, 2010, 254 p.
56.*Alberty R.**Applications of**Mathematica**.*– Wiley Press, 2011, 456 p.
57.*Shiskowski K., Frinkle K.**Principles of Linear Algebra with**Mathematica**.*–

Wiley , ISBN 9780470637951, 2011, 616 p.
58.*Kilian A.**Programmieren mit Wolfram**Mathematica**.*– Springer, 2010.
59.*Hollis S.**CalcLabs with**Mathematica**for Multivariable Calculus.*– Brooks/ Cole, ISBN–13: 9780840058133, ISBN–10: 0840058136, 2012, 274 p.
60.*Annong Xu.**Introduction to Scientific Computing:Numerical Analysis With **Mathematica**.*– China Machine Press, 2010, ISBN 9787111310914.
61.*Core Language:**Tutorial Collection.*– Wolfram Research Inc., 2008, 358 p.
62.*Hastings K.J.**Introduction to Probability with**Mathematica**.*– CRC Press, 2009, ISBN 9781420079388, 465 p.
63.*Wellin P.R.**Programming with**Mathematica:**An Introduction*, 2013.
64.*Koberlein B., Meisel D.**Astrophysics through Computation:With **Mathematica**Support*, ISBN 9781107010741, 2013.
65.*Boccara N.**Essentials of**Mathematica:**With Applications to Mathematics and Physics.*– Springer, ISBN 9780387495132, 2007.
66.*Shifrin L. Mathematica**Programming:An Advanced Introduction.*– Brunel University, 2008,***http://www.mathprogramming-intro.org/2008.***
67.*Wagon S. Mathematica®**in Action:Problem Solving Through Visualization and Computation, 3rd* ed., 2010, 574 p., ISBN 9780387754772.
68.*Bunker G. Mathematica**Quickstart.*– Illinois Inst. of Technology, 2010.
69.*Mathematica 9**documentation center:complete reference for**Mathematica 9, http://reference.wolfram.com/mathematica/guide/Mathematica.html.***
70.*http://www.haskell.org* – Web-site concerning functional programming
71.*Wadler P.* Why no one uses functional languages// ACM Notices, 1998.
72.*The fourth international seminar and international school«Mathematical and computer modeling of fundamental objects and phenomena in systems of computer mathematics»/Ed.Prof.**Yu. G. Ignat'ev.**–* Kazan: Kazan Univ. Press, 2014, ISBN 9785000193082, 126 p.
73.*The international scientifically–practical conference**ITES–2014**/Ed.Prof.**Yu. G. Ignat'ev.**–* Kazan: Foliant Press, 2014, ISBN 9785905576409, 298 p.

Monographs, textbooks, books and papers on Computer Science, Theory of General Statistics, Cellular Automata Theory and Computer Mathematics Systems, prepared and published by members of the Baltic Branch during 1995 – 2015 (Publications are grouped according to their primary purpose)

Classical Cellular Automata (Homogeneous Structures)

1. Aladjev V.Z., Hunt Ü.J., Shishakov M.L. *Questions of Mathematical Theory of the Classical Homogeneous Structures (Cellular Automata)*.– Gomel: BELGUT Press, 1996, 151 p., ISBN 5063560785*(in Russian with English summary)* ***2. Aladjev V.Z.,Hunt Ü., Shishakov M.L.****Mathematical Theory of the Classical Homogeneous Structures (Cellular Automata)*.– Tallinn–Gomel: TRG& VASCO & Salcombe Eesti Ltd., 1998, 300 p., ISBN 9063560789*(in Russian with extended English summary)*
3. Aladjev V.Z., Boiko V.K., Rovba E.A.*Classical Homogeneous Structures: Theory and Applications*.– Belarus: Grodno: Grsu, Tallinn: International Academy of Noosphere, 2008, 488 p., ISBN 9789855150207, 9789985950845 *(in Russian with extended English summary)*
4. Aladjev V.Z.*Classical Homogeneous Structures:Cellular Automata*.– USA: Palo Alto: Fultus Corporation, 2009, 536 p., 159682137X*(in Russian)* ***5. Aladjev V.Z.****Classical Homogeneous Structures:Cellular Automata*.– USA: Palo Alto: Fultus Corporation, 2009, 536 p., Adobe Acrobat eBook*(pdf)*, ISBN 9781596821385*(in Russian and English)*
6. Aladjev V.Z., Grinn D.S., Vaganov V.A.*Classical Homogeneous Structures: Mathematical Theory and Applications*.– Ukraine: Kherson: Oldi–Plus Press, 2014, ISBN 9789662890358, 520 p.
7. Aladjev V.Z.*Classical Cellular Automata:Mathematical Theory and Applica– tions*.– Germany: Saarbrücken: Scholar`s Press, 2014, ISBN–10: 3639713451, ISBN–13: 9783639713459, EAN: 9783639713459, 520 p.

General Statistics

8. Aladjev V.Z., Veetõusme R.A., Hunt Ü.J. *General Theory of Statistics:Text– book*.– Tallinn: TRG & SALCOMBE Eesti Ltd., 1995, 201 p., ISBN 1995146428 *(in Russian with extended English summary)*

9. Aladjev V.Z., Hunt Ü.J., Shishakov M.L. *Course of General Theory of Statis– tics:Textbook*. Belarus: Gomel: BELGUT Press, 1995, 201 p., ISBN 1995146429 *(in Russian with extended English summary)*
10. Aladjev V.Z.*Interactive Course of General Theory of Statistics*.– Tallinn: International Academy of Noosphere, the Baltic Branch, 2001, CD with Booklet, ISBN 9985608666*(in Russian with extended English summary)* ***11. Aladjev V.Z., Haritonov V.N.****General Theory of Statistics*. USA: Palo Alto: Fultus Corporation, 2004, 256 p., ISBN 1596820128.
12. Aladjev V.Z., Haritonov V.N.*General Theory of Statistics*. USA: Palo Alto: Fultus Corporation, 2004, Adobe Acrobat eBook, ISBN 1596820160. ***13. Aladjev V.Z., Haritonov V.N.****General Theory of Statistics*. USA: Palo Alto: Fultus Corporation, 2006, 256 p., ISBN 1596820861, Adobe Acrobat eBook *(pdf)*, ISBN 1596820810*(in Russian with extended English summary)* ***14. Aladjev V.Z., Vaganov V.A.****General Statistics*.– Tallinn: International Academy of Noosphere, the Baltic Branch, eBook,*(pdf)*, 2014, 259 p., ISBN 9789985950876.

Computer Mathematical Systems

15 Aladjev V.Z., Hunt Ü.J., Shishakov M. *Mathematics on Personal Computer*.Belarus: Gomel: BELGUT Press, 1996, 498 p., ISBN 34206140233*(in Russian with extended*

English summary)
16 Aladjev V.Z., Shishakov M.L.*Introduction into Mathematical Package* ***Mathematica 2.2.–*** Moscow**:** Filin Press, 1997, 363 p., ISBN 5895680046*(in Russian with extended English summary)*
17. Aladjev V.Z., Vaganov V.A., Hunt Ü.J., Shishakov M.L.*Introduction into Environment of Mathematical Package****Maple V.–*** Belarus**:** Minsk**:** International Academy of Noosphere, the Baltic Branch, 1998, 452 p., ISBN 1406425698*(in Russian with extended English summary)*
18 Aladjev V.Z., Vaganov V.A., Hunt Ü.J., Shishakov M.L.*Programming in Environment of Mathematical Package****Maple V.–*** Minsk–Moscow**:** Russian Ecology Academy, 1999, 470 p., ISBN 4101212982*(in Russian with extended English summary)*
19. Aladjev V.Z., Bogdevicius M.A.*Solution of Physical, Technical and Mathematical Problems with****Maple V.–*** Tallinn–Vilnius, TRG, 1999, 686 p., ISBN 9986053986*(in Russian with extended English summary)*

20. Aladjev V.Z., Vaganov V.A., Hunt Ü.J., Shishakov M.L. *Workstation for Mathematician****.–*** Tallinn–Minsk–Moscow**:** Russian Academy of Natural Sciences, 1999, 608 p., ISBN 3420614023*(in Russian with English summary)* ***21. Aladjev V.Z., Shishakov M.L.****Workstation of Mathematician****.–*** Moscow**:** Laboratory of Basic Knowledge, 2000, 752 p.+ CD, ISBN 5932080523*(in Russian with extended English summary)*
22. Aladjev V.Z., Bogdevicius M.A. Maple 6:*Solution of Mathematical****,*** *Statistical****,****Engineering and Physical Problems****.–*** Moscow**:** Laboratory of Basic Knowledge, 2001, 850 p.+ CD, ISBN 593308085X*(in Russian with extended English summary)*
23. Aladjev V.Z., Bogdevicius M.A.*Special Questions of Operation in Environment of the Mathematical Package****Maple.–*** Vilnius**:** International Academy of Noosphere, the Baltic Branch & Vilnius Gediminas Technical University, 2001, 208 p.+ CD with Library, ISBN 9985927729*(in Russian with extended English summary)*
24. Aladjev V.Z., Bogdevicius M.A.*Interactive****Maple:****Solution of Statistical****,*** *Mathematical****,****Engineering and Physical Problems****.–*** Tallinn**:** International Academy of Noosphere, the Baltic Branch, 2001–2002, ISBN 9985927710. ***25. Aladjev V.Z., Vaganov V.A., Grishin E.P.****Additional Software of Mathema– tical Package****Maple****of releases****6****and****7.–*** Tallinn**:** International Academy of Noosphere, the Baltic Branch, 2002, 314 p.+ CD with Library, ISBN 9985– 9277–3–7*(in Russian with extended English summary)*
26. Aladjev V.Z.*Effective Operation in Mathematical Package Maple****.–*** Moscow**:** Laboratory of Basic Knowledge, 2002, 334 p.+ CD, ISBN 593208118X*(in Russian with extended English summary)*
27. Aladjev V.Z., Liopo V.A., Nikitin A.V.*Mathematical Package****Maple****in Physical Modeling****.–*** Grodno: Grodno State University, 2002, 416 p., ISBN 3093318313*(in Russian with extended English summary)*
28. Aladjev V.Z., Vaganov V.A.*Computer Algebra System****Maple:****A New Software Library****.–*** Tallinn: International Academy of Noosphere, the Baltic Branch, 2002, CD with Booklet, ISBN 9985927753.
29. Aladjev V.Z., Bogdevicius M.A., Prentkovskis O.V.*A New Software for Mathematical Package****Maple****of releases****6, 7****and****8.*** Vilnius**:** Vilnius Gediminas Technical University and International Academy of Noosphere, the Baltic Branch, 2002, 404 p., ISBN 9985927745,

9986055652.

30. Aladjev V.Z., Vaganov V.A.** Systems of Computer Algebra:A New Software Toolbox for**Maple.– Tallinn: International Academy of Noosphere, the Baltic Branch, 2003, 270 p.+ CD, ISBN 9985927761.
31. Aladjev V.Z., Bogdevicius M., Vaganov V.A.**Systems of Computer Algebra: A New Software Toolbox for**Maple.**Second edition.–* Tallinn: International Academy of Noosphere, the Baltic Branch, 2004, 462 p., ISBN 9985927788. ***32. Aladjev V.Z.**Computer Algebra Systems:A New Software Toolbox for **Maple.– USA: Palo Alto: Fultus Corporation, 2004, 575 p., ISBN 1596820004. ***33. Aladjev V.Z.**Computer Algebra Systems:A new software toolbox for**Maple.–*** USA: Palo Alto: Fultus Corp., 2004, Adobe Acrobat eBook, ISBN 1596820152 ***34. Aladjev V.Z., Bogdevicius M.A. Maple:**Programming**,**Physical and Engineering Problems.–* USA: Palo Alto: Fultus Corporation, 2006, 404 p., ISBN 1596820802, Adobe Acrobat eBook*(pdf)*, ISBN 1596820810. ***35. Aladjev V.Z.**Computer Algebra Systems.**Maple:**Art of Programming.–* Moscow: BINOM Press, 2006, 792 pp., ISBN 5932081899*(in Russian with extended English summary)*
***36. Aladjev V.Z.**Foundations of programming in**Maple:**Textbook.–* Tallinn: International Academy of Noosphere, 2006, 300 p.,*(pdf)*, ISBN 998595081X, 9789985950814*(in Russian with extended English summary)*
***37. Aladjev V.Z., Boiko V.K., Rovba E.A.**Programming and applications elaboration in**Maple:**Monograph.–* Belarus: Grodno: Grsu, Tallinn: International Academy of Noosphere, 2007, 456 p., ISBN 9789854178912, ISBN 9789985950821*(in Russian with extended English summary)* ***38. Aladjev V.Z., Vaganov V.**Modular programming: **Mathematica**vs**Maple,** and vice versa.–* USA, CA: Palo Alto: Fultus Corporation, 2011, ISBN 9781596822689, 418 p.
***39. Aladjev V.Z., Bezrukavyi A., Haritonov V.N., Hodakov V.**Programming: System**Maple**or**Mathematica?**–* Ukraine: Kherson, Oldi–Plus Press, 2011, ISBN 9789662393460, 474 p.*(in Russian with extended English summary)* ***40. Aladjev V.Z., Boiko V.K., Rovba E.**Programming in system**Mathematica** and**Maple:**A Comparative Aspect.*Belarus: Grodno, Grodno State University, 2011, 517 p.*(in Russian with extended English summary)*
41. Aladjev V.Z., Grinn D.S., Vaganov V.A.**The extended functional means for system**Mathematica.– Ukraine: Kherson: Oldi–Plus Press, 2012.

42. Aladjev V.Z., Grinn D.S.** Extension of functional environment of system **Mathematica.– Ukraine: Kherson: Oldi–Plus Press, 2012, ISBN 978–966–2393–72–9, 552 p.*(in Russian with extended English summary)*
43. Aladjev V.Z., Grinn D.S., Vaganov V.A.**The selected system problems in software environment of system**Mathematica.– Ukraine: Kherson: Oldi–Plus Press, 2013, ISBN 9789662393729, 556 p.*(in Russian with English summary)*
***44. Aladjev V.Z., Vaganov V.A.**Extension of the**Mathematica**system functionality.–* Estonia: Tallinn, TRG Press, 2015, ISBN 9789985950883**,** 563 p.

Computer Science

***45. Aladjev V.Z., Hunt Ü.J., Shishakov M.L.** Basics of Computer Informatics: Textbook.–*

Tallinn–Gomel: Russian Academy of Noosphere & TRG, 1997,
396 p., ISBN 5140642545*(in Russian with extended English summary)*
46. Aladjev V.Z., Hunt Ü.J., Shishakov M.L.*Basics of Computer Informatics: Textbook.–* Moscow, Filin Press, 1998, 496 p., ISBN 5895680682*(in Russian with extended English summary)*
47. Aladjev V.Z., Hunt Ü.J., Shishakov M.L.*Basics of Computer Informatics: Textbook, Second edition.–* Moscow: Filin Press, 1999, 545 p.*(in Russian with extended English summary)*

Scientific Reports and Collection of Papers

48. Aladjev V.Z., Hunt Ü.J., Shishakov M.L. *Scientific–research Activity of the Tallinn Research Group:Scientific Report over a period 1995–1998.–* Tallinn– Gomel– Moscow: TRG & VASCO, 1998, 80 p., ISBN 1406429856*(in Russian with extended English summary)*
49. Aladjev V.Z. et al.*Electronic Library of Books and Software for Scientists, Experts,Teachers and Students in Natural and Social Sciences.–* USA: Palo Alto: Fultus Corporation, 2005, CD, ISBN 1596820136*(in Russian and English)* ***50. Aladjev V.Z.****Modular programming:* ***Maple****or****Mathematica*** *–A subjective standpoint /Intern. school«Mathematical and computer modeling of fundamental objects and phenomena in systems of computer mathematics»,* ed.***Y. G. Ignat'ev.–*** Kazan: Kazan University Press, 2014, pp. 18–32.

About the Authors

Professor ***Aladjev V.Z.*** was born on*June 14,1942* in the town*Grodno (West Byelorussia).* Now, he is the First vice–president of the*International Academy of Noosphere (****IAN****)*, and academician–secretary of Baltic branch of the***IAN*** whose scientific results have received international recognition, first, in the field of Cellular Automata theory.***Aladjev V.Z.*** is known for the works on computer mathematical systems too. He is full member of a number of the Russian and International Academies. Prof. Dr.***AladjevV.Z.*** is the author of more than***500*** scientific publications, including***90*** books and monographs, published in many countries. He participates as a member of the organizing committee and/or a guest lecturer in many international scientific forums in mathematics and cybernetics. In*May,2015* Prof.***Aladjev V.Z.*** was awarded by*Gold* medal***"European Quality"*** of the European scientific and industrial consortium***(ESIC)*** for works of scientific and applied character.

Dr. ***Vaganov V.A.*** was born on*February 2,1946* in*Primorye Territory* (*Russia).* Now***Vaganov V.A.*** is the proprietor of the firms*Fortex* and*Sinfex,* engaging of problems of delivery of industrial materials to the firms of the Estonian republic. Simultaneously***V.A. Vaganov*** is the executive director of the Baltic branch of the***IAN.Vaganov V.A.*** is known enough for the investigations on automation of economical and statistical works. Result was a series of the scientifical and applied works published in Republican editions and at All– Union conferences. Dr.***Vaganov V.A.*** is the honorary member of the***IAN*** and the author of more than***60*** scientific publications, including***10*** books.