The Basis System, part 1

The Basis Development Team

November 13, 2007

Lawrence Livermore National Laboratory
Email: basis-devel@lists.llnl.gov

UCRL-MA-118543

CONTENTS

CHAPTER

ONE

The Basis System

1.1 Environment Variables

Before using Basis, you should set some environment variables as follows.

- `BASIS_ROOT` should contain the name of the root of your Basis installation, `/usr/apps/basis` for example.
- `MANPATH` should contain a component `$BASIS_ROOT/man`.
- Your path should contain a component `$BASIS_ROOT/bin`.
- `DISPLAY` should contain the name of your X-Windows display, if you will be doing X-window plotting.
- `NCARG_ROOT` should contain the name of the root directory of your NCAR 4.0.1 or later distribution, if you have it.

Check with your System Manager for the exact specifications on your local systems.

1.2 Basis Is Both a Program and a Development System

Basis is a system for developing interactive computer programs in Fortran, with some support for C and C++ as well. Using Basis you can create a program that has a sophisticated programming language as its user interface so that the user can set, calculate with, and plot, all the major variables in the program. The program author writes only the scientific part of the program; Basis supplies an environment in which to exercise that scientific programming, which includes an interactive language, an interpreter, graphics, terminal logs, error recovery, macros, saving and retrieving variables, formatted I/O, and on-line documentation.

`basis` is the name of the program which results from loading the Basis System with no attached physics. It is a useful program for interactive calculations and graphics. Authors create other programs by specifying one or more packages of variables and modules to be loaded. A package is specified using a Fortran source and a variable description file in which the user specifies the common blocks to be used in the Fortran source and the functions or subroutines that are to be callable from the interactive language parser.

Basis programs are *steerable applications*, that is, applications whose behavior can be greatly modified by their users. Basis also contains optional facilities to help authors do their jobs more easily. A library of Basis packages is available that can be added to a program in a few seconds. The programmable nature of the application simplifies testing and debugging.

The Basis Language includes variable and function declarations, graphics, several looping and conditional control structures, array syntax, operators for multiplication, dot product, transpose, array or character concatenation, and a stream I/O facility. Data types include real, double, integer, complex, logical, character, chameleon, and structure. There are more than 100 built-in functions, including all the Fortran intrinsics.

Basis' interaction with compiled routines is particularly powerful. When calling a compiled routine from the interactive language, Basis verifies the number of arguments and coerces the types of the actual arguments to match those expected by the function. A compiled function can also call a user-defined function passing arguments through common.

1.3 About This Manual

The Basis manual is presented in several parts:

I. Running a Basis Program, A Tutorial

II. Basis Language Reference

III. EZN User Manual: The Basis Graphics Package

IV. The EZD Interface

V. Writing Basis Programs: A Manual For Program Authors

VI. The Basis Package Library

VII. MPPL Reference Manual

The first three parts form a basic document set for a user of programs written with Basis. The remainder form a document set for an author of such programs.

Basis is available on most Unix and Unix-variant platforms. It is not available for Windows or Macintosh operating systems.

A great many people have helped create Basis and its documentation. The original author was Paul Dubois. Other major contributors, in alphabetical order, have been Robyn Allsman, Kelly Barrett, Cathleen Benedetti, Stewart Brown, Lee Busby, Yu-Hsing Chiu, Jim Crotinger, Barbara Dubois, Fred Fritsch, David Kershaw, Bruce Langdon, Zane Motteler, Jeff Painter, David Sinck, Allan Springer, Bert Still, Janet Takemoto, Lee Taylor, Susan Taylor, Peter Willmann, and Sharon Wilson. The authors of this manual stand as representative of their efforts and those of a much larger number of additional contributors.

Send any comments about these documents to ”basis-devel@lists.llnl.gov” on the Internet.

CHAPTER

TWO

Getting Started

2.1 What is Basis?

Basis is two things: It is a system used to produce computer programs, and it is the name of a programming language which serves as the user interface to a program so produced. To say it another way, an author uses the Basis System to make a program named **foo**, and the user uses the Basis Language to write the input file for **foo**. Just to add to the confusion, one such **foo** is a program named **basis**, which consists of nothing but an interpreter for the Basis Language.

The purpose of this manual is to give you a quick introduction to working with the Basis Language, so that you can get going as rapidly as possible. This tutorial is not a complete description of the Basis Language, but is intended to build enough of a foundation that you can later learn more sophisticated features from the full reference manual. From now on, when we say Basis, we mean the Basis Language, not the Basis System that the author used to build **foo**. When we say **basis**, we mean the program basis which you can use as a practice vehicle for learning the language, or as a useful interactive calculator and plotter.

The Basis Language looks very much like Fortran, so if you know Fortran, you should be able to pick up the elements very quickly. Unlike Fortran, though, Basis is an interpreted language, which means that (usually) Basis statements are executed as soon as they are typed in. Basis contains a lot of built-in functions, input-output facilities, and can interact with compiled code and variables. You can even compute Basis' input with Basis itself.

2.2 Starting the Program

If you are using basis by itself, you simply type in

```
basis
```

on the computer of your choice. You may be using some other code (Lasnex is an example) which consists of Basis and a lot of other compiled code, in which case you may type in some other name.

Basis (or whatever) will initialize and when the process is complete, it will print out a prompt for you. The usual default prompt is

```
Basis>
```

although this is one of the many customizable features of Basis that can be changed by the program author. For sure, though, you will know when you are being prompted. Upon receiving the prompt, you can immediately begin typing in Basis statements.

The default for a program built with the Basis System is that any arguments on the command line are simply treated as the first line of input. However, this is one of those things an author may have changed, so check your specific program's documentation.

2.3 Getting Information

The key command for finding your way around a Basis program is the `LIST` command. Basis commands such as `LIST` may be entered either in all lower case or all upper case. Sometimes we will use upper case to emphasize that the word is a Basis reserved word, but usually people enter them in lower case.

If you enter the command

```
list
```

you will get output something like the following:

```
list options
------------
list                   [print the list options]
list par.Attributes
list [pkg.]functions
list Groupname
list [pkg.]groups
list idname
list macros
list packages
list [pkg.]variables
NOTE:Groupname is the name of a group in any package on the
     current search stack.
NOTE:Groupname can be abbreviated.
NOTE:idname is the name of a function, macro, or variable in any
     package on the current search stack.
NOTE:list groups, functions, and variables list local and user
     created groups, functions and variables respectively
     unless pkg. is utilized.
NOTE:list pkg.functions lists the built-in and compiled functions
     in the database for that package.
```

You can use the `LIST` command to get all sorts of information about Basis functions, predefined macros, constants and variables, and the like. Enter:

```
list packages
```

and Basis will list the packages that are loaded. One of these is *par*, the Basis parser package. Now enter

```
list par.groups
```

and you will see a list of the groups that make up this package. Now you can enter `LIST` followed by one of the group names (or a prefix of it) and you will see an explanatory list of the items in that group. You can ask to have an individual item listed (such as a compiled or built-in function) to get more information about that item (for instance, its parameters, what it does, what type it returns (if any), and so on).

2.4 Comparison of Basis and Fortran

We summarize here very briefly the important similarities and differences between Basis and Fortran. In each section of this manual we will summarize further similarities or differences pertinent to the topic under discussion. For the real details you will need to go to the reference manual.

2.4.1 Major Similarities between Basis and Fortran

1. Basis has pretty much all of the Fortran operators and delimiters you are familiar with, and they have the same functions and precedences. It has many more operators, but you won't need them when you're getting started.
2. Basis expressions (including array references and function invocations) look just like they do in Fortran.
3. Basis has all the data types available in Fortran (and more).
4. Basis `IF` statements look just like Fortran `IF` statements.
5. Basis `DO` statements are very similar to Fortran `DO` statements, but have no label and end in an `ENDDO` statement.

2.4.2 Major Differences between Basis and Fortran

Mostly, Basis will do what an experienced Fortran programmer expects. The chief incompatibility is in the form of the input. In general, Basis extends the ideas of Fortran to an array-syntax interpretive environment.

1. Basis is interpreted rather than compiled.
2. Basis comments start with # and extend to the end of the line.
3. Basis has no statement numbers or goto's.
4. Basis input is essentially free-form (columns are not significant). There is no continuation column; a statement is continued from one line to the next by ending the line with
   or a comma, open paren or bracket, or operator.
5. There are no default types in Basis; variables must be declared.
6. Basis function names and formal arguments must NOT be typed.
7. Basis functions may return virtually any kind of entity, including arrays.
8. Basis passes actual parameters to functions by value (i.e., as copies), not by reference (i.e., as addresses).
9. Several Basis statements can appear on the same line, if separated by semicolons(;).
10. Basis is case sensitive, that is, upper and lower case are distinguished. Basis reserved words may be entered either in all caps or all lower case, though.
11. Spaces are significant in Basis (except in quoted strings and comments, of course), and act as delimiters between tokens.
12. Double quotes are used for strings – single quotes are used for something else.

CHAPTER

THREE

The Basis Language

3.1 Assignments and Expressions

One of the first things you are going to want to do is assign values to variables. These may be variables which are already built into Basis, but could also be variables in compiled code which the author has made available to the Basis. In a future chapter, we'll show you how to declare your own Basis variables, and then, of course, you can assign values to these as well. All variables to which some quantity is assigned must have been declared previously, either by Basis, by you, or by the program author.

Here are some examples of assignments. These are all assignments to variables which are predeclared in Basis. These examples are designed for you to follow along with on a terminal.

```
debug = yes
fuzz = 9
switches(2) = pi
switches(5) = 2.0 * cos ( switches(2) / 3 )
```

yes is a predefined constant in Basis (value 1), and *pi* is a predefined famous number. *debug* is a predeclared variable which, if set to *yes*, toggles on more detailed debugging diagnostics. We recommend '`debug=yes`' for beginners. *fuzz* controls the accuracy of real numbers printed out by Basis (it is the number of digits after the decimal point). *switches* is a predeclared real scratch array which we have used here to show assignment to an array element. The last assignment illustrates an arithmetic statement used in an assignment; its meaning should be obvious to any user of Fortran. `cos` is only one of a very large number of built-in functions available in Basis. All the usual ones are available; see the reference manual for complete details, or use the `LIST` command to find out more.

For your convenience, Basis has predefined the following constants: *yes = 1, no = 0, on = "on", off = "off", true = logical true,* and *false = logical false*. In many cases `true` and `yes` can be used interchangeably, as can `false` and `no`.

If you want to print out the values of some of the above variables, simply type in a comma-delimited list of their names, for example

```
fuzz , switches(2)
```

and Basis will print out the values (if real, to the number of digits specified by *fuzz*). In fact, you can type in any expression or list of expressions, and Basis will compute it and print out its value (if it can). For instance, try typing in

```
2.0 * cos ( switches(2) / 3 ), x + y + 5
```

You should get an “Unknown variable” message from the second expression. When Basis encounters an error, it stops processing the current line, writes a diagnostic to the terminal and some debugging information to a trace file (more if *debug* is *yes*, as mentioned above), and returns to the prompt. It is now ready to accept further statements.

Try typing in some sort of declaration for *x* and *y*, such as

```
integer x , y
```

and then assign something to them, then type the expression again. This time you will get its computed value. At this point, assuming you know Fortran, you should be able to type in various Fortran-compatible declarations, expressions, and assignment statements, to get more of a feel for Basis. Don’t worry about making mistakes; Basis is very tolerant of them and will come back again and again.

You might also want to experiment with the predeclared Basis chameleon variables *$a*, *$b*, ... , *$z*. These variables exist in Basis but initially have no type or value; they get their types and values by being assigned to, and this can be done again and again. They are thus called chameleons for the obvious reason that they “change color” to “blend into their environment”, i.e., change type and value to whatever is assigned to them. Try the following sequence of statements:

```
$b
$b = cos ( pi / 3.0 )
$b
$b = 6
$b
```

The first statement will cause an error because, although *$b* exists, it is undefined. The second statement assigns *$b* a real value, which the third prints. The fourth assigns it an integer value, which the fifth prints. You might want to experiment a bit with the chameleons before proceeding.

Basis allows logical variables and logical-valued expressions. As in Fortran, such expressions would normally be used in an `IF` statement to control some execution choice. However, logical values can be assigned too.

```
integer x, y, m
x = 3
```

```
y = 5
logical z
z = x > y
z
```

Basis will tell you that the value of *z* is `false`. You can use either $>$ or the more Fortran-like `.gt.` for the comparison operator. If you do use `.gt.` make sure the periods are not ambiguous. For example, '`3.gt.y`' is not going to work right but '`3 .gt.y`' is ok.

3.2 Input from a File

So far we've discussed input from the terminal, at the Basis prompt. Often, however, we input statements from a file. If you have code in a file named 'my_funcs' then you can have it read in and interpreted by entering:

```
read my_funcs # no quotes necessary here
```

A file that you are reading in can itself contain `read` commands, and so on, up to a depth of twenty. If Basis detects an error in a file that it is reading in, it will close the file and give a diagnostic, then return to the Basis prompt for input. A `resume` statement will begin reading that file again at the line that failed (assuming you snuck off and fixed it, and the line was at an appropriate place to resume input).

You can execute commands using the Bourne shell from within Basis (like starting up your editor in order to fix the input file) by beginning the line with an exclamation point:

```
!emacs my_funcs
```

You will return to Basis when the command exits.

3.3 Some Differences from Fortran

There is a lot more to Basis expressions than just imitating Fortran. Basis operators are more general, and there are more of them.

1. Most Basis operators are more general than their Fortran counterparts. For instance, '`*`' and other arithmetic operators will perform component-wise operations on vector or array arguments.

2. Basis has many additional operators such as matrix multiply and dot product; the operator '`//`' which concatenates strings and arrays.

3. Unlike Fortran, the Basis logical and relational operators also have symbolic versions; for instance '&' for `.and.`; '`~=`' or '<>' for '`.ne.`', and the like.

We'll cover more of this as we proceed.

3.4 Declaring Variables

The examples in this chapter are designed to be executed as you read them. For brevity, variables declared in an earlier example are frequently reused without redeclaring them in later examples.

The usual Fortran types are available for declaring variables, such as:

```
INTEGER x, y, z
REAL i, j, k = 2.0
DOUBLE d = 2.d0
COMPLEX c = 2.0 + 3.0i
LOGICAL l1 = true, l2 = false
CHARACTER*3 ch = "abc"
```

Note the following (mostly minor) differences from Fortran:

- Variables can be initialized in their declarations (as `k`, `d`, `c`, `l1`, `l2`, and `ch` above). These initialization expressions need not be just constants, but can be arbitrary expressions, as long as all values in them are known when the statement is encountered. Even '`real x=1, y=x`' is ok.
- The use of `DOUBLE` alone, not `DOUBLE PRECISION`.
- The notation for imaginary constants (a numerical quantity followed by `i`, with no space between).
- `true` and `false` without the surrounding periods as in Fortran.
- Variables which are not explicitly initialized are set to 0, or to blanks if they are of character type.

Declare array variables of up to seven dimensions as follows:

```
REAL x(10), y(3,5), z(-3:5, 7:10)
```

The lowest value of the subscript range defaults to 1 unless a different value is specified before a colon, as in `z` above. Thus, `x` is subscripted $1 \ldots 10$, `y` from $1 \ldots 3$ and $1 \ldots 5$, and `z` from $-3 \ldots 5$ and $7 \ldots 10$. An individual array can be initialized by a vector of values that follows its type declaration:

```
INTEGER i(10) = [0,0,0,0,0,1,1,1,1,1], j(5) = [1,2,3,4,5]
```

The vector components may be arbitrary expressions.

3.5 Some Elements of Array Syntax

Basis operators support arithmetic on arrays of arbitrary size and shape, except that binary operations (`*`, `/`, etc.) require their operands to be compatible in size and shape; and assignments require that the object being assigned must be storable as a subobject of the receiving item. For all the details on this subject the reader is referred to the Basis reference manual.

Array operands can be expressed in various ways. An entire array is specified by its name without subscripts: Thus, in the example above, either `i` or `i()` refers to the entire array of ten elements as specified. One can also use subscript range notation to extract subarrays of a given array.

```
     i(1:5) # will be the first five elements of i
     i(2:10:2) # will be [i(2), i(4), i(6), i(8), i(10)]
     i(3:5)+j(1:3) # will be [1,2,3]
```

A range specification consists of

```
low_dimension:high_dimension:step_size
```

`step_size`, if omitted, defaults to 1. `low_dimension` and `high_dimension` must be within the declared range, and if either is omitted, defaults to the declared value. Aside: These numbers must be integers. The range notation with a real component means a vector of real numbers. Try entering

```
0.:1.:10   #vector of ten reals from 0. to 1.
0.:1.:.02  #From 0. to 1. in steps of .02
```

By contrast, if you enter `0:10:2` you are printing out a range and such a range can be used as a subscript.

You can apply the usual binary operators to objects of the same size and shape (regardless of subscript values), and the operation will be applied to each component. The only exception to this compatibility requirement is that scalars may participate in operations with arrays, in which case the scalar is applied to each element of the array. For example,

```
i + 5 # adds 5 to each component of i
```

The same rules apply to arrays with more than one dimension. There is an additional rule applying when subscripts are missing. A missing subscript will always default to its minimum value (as declared), except when all are missing, which means the entire array. Examples:

```
integer a(5,5)
a()          # is the entire 5 by 5 array
a(5)         # is just a(5,1)
a(1:5)       # is [a(1,1),a(2,1),a(3,1),a(4,1),a(5,1)]
a(1:5) + j   # valid operation since j is the same size and shape
```

3.6 IF Statements

Basis `IF` statements are exactly like Fortran, except that it is possible (and preferable, we think) to use these comparison operators:

- > instead of `.gt.`
- >= instead of `.ge.`
- < instead of `.lt.`
- <= instead of `.le.`
- = or == instead of `.eq.`
- <> or ˜= instead of `.ne.`
- ˜ instead of `.not.`

The following will determine the maximum of two numbers:

```
 if ( x > y ) then
   m = x
 else
   m = y
endif
```

Unlike Fortran, you do not use variant forms of the comparison operators for non-numeric types. Most Fortran programmers are blissfully unaware of `.eqv.`, but if you know about it, forget it.

3.7 Looping Constructs

Basis has several looping constructs. The most-used one is a `DO/ENDDO` statement that is close to the Fortran `DO`.

Suppose, for example, that *a*, *b*, and *c* are all *n* by *n* square matrices, and that we want to put the matrix product of *b* and *c* into *a*. This could be done by the following:

```
integer i1, i2, i3, n = 5
integer a(n,n), b(n,n), c(n,n)
b = b + 1
c = c + 2      # setting values for b and c.
do i1 = 1 , n
   do i2 = 1 , n
      a(i1,i2) = 0
      do i3 = 1 , n
         a(i1,i2) = a(i1,i2) + b(i1,i3) * c(i3,i2)
      enddo
   enddo
enddo
```

The only real difference between this statement and the Fortran `DO` is that Basis does not have statement labels so the do-loops are delineated by the `DO` ... `ENDDO` pair. Since there is no statement number, none appears after the reserved word `DO`. As in Fortran, an increment can be specified, but if not, it defaults to 1.

3.8 Vector Syntax

We can greatly increase the speed of array calculations by using array syntax where possible. We can rewrite the matrix multiply as:

```
do i1 = 1 , n
   do i2 = 1 , n
      a(i1,i2) = sum(b(i1,) * c(,i2))
   enddo
enddo
```

or use the dot-product operator `!`:

```
do i1 = 1 , n
   do i2 = 1 , n
      a(i1,i2) = b(i1,) !c(,i2)
   enddo
enddo
```

To really make it easy, use the matrix-multiply operator `*!`:

```
a = b  *! c
```

Other matrix facilities include `transpose(a)`, and concatenation (`//`). The latter operation appends one array to the end of another, forming a one-dimensional object whose size is the total number of elements of the two components. With the preceding declarations of *i* and *j*, you might want to try

```
i//j
i + j // j
```

to see what happens, and see if you understand why.

Square brackets are used for array building notation. Up to now, we have simply shown them used for literal arrays, but you might want to experiment with them to see what they can do:

```
[[1,2],[3,4]]  # The matrix
               # 1 3
               # 2 4
[j,j+1]        #  = [[1,2,3,4,5],[2,3,4,5,6]]
[j,2,3]        # [1,2,3,4,5,2,3]
[[j,j+1],99]   # [1,2,3,4,5,2,3,4,5,6,99]
[[j,j+1],[j-1]]# [[1,2,3,4,5],[2,3,4,5,6],[0,1,2,3,4]]
```

The final tool for building arrays is the `:=` assignment operator, which appends the right hand side to the left hand size, thus changing its size. This is usually used to build up a list whose length is not known in advance.

```
integer mylist(1:0) #empty integer list
integer k
do k = 1, 100
   if(mod(k**2,6)==0) then
      mylist  := k
   endif
enddo
mylist
```

prints the list of those integers between 1 and 100 whose squares are divisible by 6.

3.9 Differences between Basis and Fortran

1. Basis does not have `DIMENSION` or `EQUIVALENCE` statements.
2. The Basis `CHARACTER` type does not allow the syntax '`character x*3 , y*19`'.
3. Basis words such as `INTEGER` (and other type names), `IF`, `DO`, etc., are reserved words and cannot be used as variable names.

4. Basis has additional types (`RANGE`, `INDIRECT`, and `CHAMELEON`), which are not available in Fortran.
5. In Basis, a type can be prefaced with a scope, such as a package name. The most frequently used of these is `GLOBAL`, as in ‘`global real x`’. This declaration makes *x* a global variable and therefore *x* will exist even after the return of the function in which this declaration occurs.
6. Most Basis functions (e.g., `sqrt`, `sin`,`cos`, `exp`, etc.) will accept arrays as arguments, perform the indicated function on components, and return an array of the results.
7. Basis has many more types of looping statements, such as `DO`... `UNTIL`, `FOR` (similar to the C statement), `WHILE`... `ENDWHILE`, etc., described in more detail in the reference manual.

CHAPTER

FOUR

Graphics

A Basis program may or may not have a graphics package attached. The standard package attached to **basis** is called `ezc`. The current version of `ezc` uses NCAR graphics and is sometimes referred to as `ezn` to distinguish it from an earlier, non-NCAR, version.

The graphics devices available depend on those available in the graphical kernel system (GKS) available at your site. At a minimum, this is NCAR's GKS, which produces output files called NCGM files, which can be processed by NCAR utilities **ctrans** and **idt**. Another GKS is one made by a company called ATC. The ATC-GKS has drivers for X-Windows, Postscript, Tektronics, and CGM files. These CGM files can be converted to NCGM files using the NCAR utility **cgm2ncgm**.

We will not attempt to reproduce the EZC manual here, but the following sample session may be enough to get you going.

Before using a program containing EZN, make sure you have set the Basis Environment Variables as described in the first chapter.

It assumes ATC-GKS and begins with turning on both CGM file output and an X-Window. It then plots two curves on the same graph, advances the frame, and makes a contour plot. For the contour plot, titles are added and the frame limits are controlled by the user.

```
abics[1] basis
Basis    (basis, Version 931116)
Run at 10:55:17 on 11/22/93 on the sun4 machine, suffix 18021x
Initializing Basis System
Basis 9a
Initializing PFB Interface
PFB 1.0
Initializing 3-D Surface Plotting Routine
Initializing Device Package
EZD Graphics Devices 2.1
Initializing EZCURVE/NCAR Graphics
ezn /NCAR/ATC 4.2
Basis> real x=iota(100),y1=x**2,y2=x**2.1
Basis> real xx=iota(-5:5),yy=xx+6,zz=outer(xx,yy)
Basis> ezcshow=false   #see below
Basis> cgm on
Beginning CGM File problem.001.cgm
Beginning CGM Log problem.001.cgmlog
Basis> win on
Basis> plot y1 x
Basis> plot y2 x color=red style=dashed
Basis> nf
Basis> frame -4. 4. 0. 10.
Basis> titles "Top" "Bottom" "Left" "Right"
Basis> plotz zz xx yy
Basis> end
Closed CGM File problem.001.cgm,     1 frames.
Closed CGM Log File problem.001.cgmlog
   CPU (sec)   SYS (sec)
       2.733       3.000
abics[2]  cgm2ncgm < problem.001.cgm > foo.ncgm
abics[3]  ctrans -d ps.mono foo.ncgm | lpr
```

In the last two lines, the ATC-GKS CGM file was converted to an NCGM file, and the **ctrans** utility was used to send the picture to a monochrome postscript printer. To view the file in an X-Window do

```
ctrans -d X11 foo.ncgm
```

A window will appear; click in it to see the next frame.

The '`ezcshow=false`' line causes the plots to not be displayed until all the objects have been added to them and the `nf` ("new frame") is executed. Without it three frames would have been generated because the first one would have contained the plot of *y1* and the second the plot of both curves.

'`outer(xx,yy)`' forms the outer product of the vectors *xx* and *yy*, making *zz* a matrix.

Use the `LIST` command on the `ezc` package to get more ideas about what you can control.

CHAPTER

FIVE

Text Input and Output

5.1 Stream Input

5.1.1 The $>>$ Operator

This section explains how to read numbers in from a text file, which may contain numbers in various formats as well as various non-numeric information which is to be skipped over.

The operator $>>$ is called the "stream input" operator and it is inspired by the operator in the language `C++`. It has the basic form:

```
unit >> variable
```

where `unit` is an integer which has been set as the result of calling the function `basopen("filename","r")`. The '`r`' stands for "read". The function `basclose(unit)` is used to close the input file when finished.

Suppose the input file 'testdata' looks like this:

```
c special input file
     time = 2.56 , factor = 13.51e-2
     1.2 2.3 3.4 4.5
```

Then here is some Basis code which would read in the numbers in this file:

```
integer i1 =  basopen("testdata","r")
real x,y, d(2,2)
i1 >> x
i1 >> y
i1 >> d
call  basclose(i1)
```

Then after the execution of the above sequence of instructions, `x=2.56`,`y = .1351`, `d(1,1) = 1.2`, `d(2,1) = 2.3`, `d(1,2) = 3.4`, and `d(2,2) = 4.5`. The remaining characters in the file (the "noise") will have been ignored. Note that *d* appeared in the input list with no subscripts (thus implying the entire array), and that it was read in in column major order (first subscript varying most rapidly). Basis is like Fortran, which also stores its arrays in column major order.

You should close the input file with:

```
call  basclose (unit)
```

NOTE: Files opened by `basopen` are automatically closed whenever an error occurs.

A series of stream input statements can be abbreviated by multiple stream input operators per statement. The above is equivalent to:

```
i1 >> x >> y >> d
```

You may use the terminal as an input file (but don't open or close it, please!) by using `stdin` as the unit number, or omitting the unit number.

5.1.2 Detecting end-of-file

It is the user's responsibility to determine whether the end of a file has been reached. For this reason an end-of-file flag (*eof*) has been provided. *eof* is an integer variable which contains the value `no` if the last read attempt was successful, and `yes` if the last read attempt was unsuccessful. The user should use the *eof* variable when reading input. For example, this is how one could read an array of unknown length (first create a file '`numbs`' with some numbers in it):

```
real x(1:0), y    # x starts empty
integer i1 =  basopen("numbs", "r")
eof = no          # making sure eof is no to start with
i1 >> y           # read y
while ( eof = no )
  x  := y          # append y to x
  i1 >> y
endwhile
call  basclose(i1)
```

When the end of a file is encountered, the variables that cannot be assigned new values because of lack of input retain their original values. Once `eof` is `yes` for a specific file, the user should make no further attempt to read input from that file.

`eof` always reflects the status of the last file read from. Test its status on a particular file before you issue an input command for some other file, which may change its status.

5.2 Stream Output

5.2.1 The $<<$ Operator

Stream output is very similar to stream input, which we studied in an earlier chapter. You open the file for writing:

```
unit =  basopen ( "file" , "w" )
```

You give one or more output commands, unit number first, then an expression, and as many more operator-expression pairs as desired:

```
unit << fee << fie << fo << fum
```

Output expressions can be any legal Basis expression. Each output command will start on a new line, but may or may not be more than one line long. When finished, close the file (Note this is the same call to close an input file):

```
call  basclose ( unit )
```

You won't get any spaces between the different parts of the output unless you put them there, as in

```
unit << "x is " << x << " and y is " << y
```

You may use the terminal as an output device (but please don't open or close it!) by using `stdout` as the unit number, or by omitting the unit number. This makes it easy to make comments:

```
<< "Dear Sir, your run is proceeding quite nicely."
```

You may put stream output onto your current graphics devices by using `stdplot` as a unit number; again, neither open or close stdplot. Many Basis users like to document their graphics files by using stdplot to print the values of input parameters at the start of their graphics files. You can also redirect most terminal output to the graphics files with

```
output graphics
```

with a subsequent `output tty` to restore terminal output.

5.2.2 Controlling Line Length

You can force a line break anywhere in the output by placing the reserved word `return` between any two output operators. You can cause the automatic line break after each output command to be suppressed by setting the Basis variable `autocr` to `no` (its default value is `yes`). In this case, Basis will fill an output buffer before it sends the output and a line break, unless there is a `return` somewhere along the way. (You, the user, can do nothing to alter the size of the Basis output buffer.)

5.2.3 Formatting: format

Basis contains a built-in function `format` which takes a number and some integer parameters and returns a character string. It can be used to produce output similar to Fortran formatted output. However, `format` only accepts a scalar argument, so if you want to send out an array, it will have to be in a loop where you send elements one at a time. To format an integer, use

```
format ( <integer expression> , <field width> )
```

after an output operator. This function call returns the acsii character string for the integer expression, with exactly the number of characters asked for (right justified, if necessary), except that if you specify 0, it will give you exactly as many as are necessary.

The `format` function does not accept complex numbers, by the way, so you would have to format the real and imaginary parts separately; use `float(c)` and `cmplx(c)` to extract the real and imaginary parts.

To format a real expression, use

```
format (<expression>,<field width>,<dec. places>,<EorF>)
```

after an output operator. The string returned by this call will have exactly the number of characters specified by the width, unless you ask for 0, in which case it will give you only as many as are necessary. The number will be right justified if necessary. `<dec. places>` tells how many digits you want to the right of the decimal point. If `<EorF>` is 0, it will give a Fortran `E`-type format, and if it is 1, it will give a Fortran `F`-type format.

CHAPTER

SIX

Functions

6.1 Defining Functions

Now we'll learn how to define functions in Basis. When a function is defined, it is compiled into an internal form and stored. The function will then be executed if the function is invoked in a Basis expression.

In this section, we will look at examples of functions. By the time you finish this tutorial, you should be able to write many useful functions.

The following function computes the absolute value of the difference of its arguments.

```
FUNCTION adiff(x,y)
return abs(x-y)
ENDF
```

To try it, enter `adiff(-5.,5)`. Then try `adiff( [1,2],[9,2])`.

```
list adiff
```

displays the information that Basis has stored about this function; if you answer "`y`" to the "`Dump intermediate code?`" question, you will get a hint of what the internal code of this function looks like.

Here are some notes about the function `adiff`:

1. Note that neither the function nor its formal parameters is typed. This is what permits `adiff` to return different types and shapes of results depending on the input.
2. In Basis, a value is returned from a function by using the `RETURN` statement followed by a value (similar to C), *not* by assigning a value to the function name (as Fortran would do it).
3. The function ends with reserved word `ENDF`, not `END`. The reserved word `END`, in Basis, causes Basis to terminate. It can not be legally used in any other context.

You can declare local variables inside functions, which will not exist after they return.

```
function diff(x)
# return first differences of x
chameleon z=shape(x,length(x))
return z(2:)-z(1:length(z)-1)
endf
```

Note the use of the chameleon type; this means `diff` works properly whether `x` is integer, real, double, or complex. The shape function makes sure `z` is a vector whose lowest index is 1 so that we can subscript it correctly in the following line.

6.2 Arguments Passed by Value

Basis passes a copy of each argument to the function ("pass by value") while Fortran passes the address of the argument ("pass by reference"). Thus a Fortran function which assigns a value to one of its arguments will cause a change in the value of the actual argument, while this will not occur in Basis, since only a copy is altered.

When you call a Fortran function from Basis, and the function changes one of its arguments, you must tell Basis to pass an argument by address by prefixing the name of the argument with an ampersand:

```
real x
call second(&x)
```

Here, `second` is a compiled function which returns the time used in its argument.

There are very few cases where it is necessary to have a Basis Language function change an argument, because a function can return an entire array as its value, if necessary. However, Basis has an `INDIRECT` type that allows you to pass the name of the argument and then operate on that in the function:

```
FUNCTION w(namex)
INDIRECT y=namex
y(3) = 7.
ENDF
REAL x(100)
call w("x")
```

will result in `x(3)` being set to 7. By contrast,

```
FUNCTION w(y)
y(3) = 7.   #THIS IS USELESS
ENDF
REAL x(100)
call w(x)
```

does NOT modify `x`; rather, a copy of `x` has been modified, and then discarded when `w` returned.

In other words, the name of an argument whose value is to be changed should be passed to the function as a character string. Within the function, a local variable is declared `INDIRECT` and initialized to the name of the formal parameter to be changed. Then assignment to the `INDIRECT` variable will result in changing the actual argument in the calling routine.

6.3 Further Differences with Fortran

1. Basis does not have the `SUBROUTINE` declaration; a Basis function can return a value or not.
2. `COMMON` variables do not exist in Basis.
3. Globally accessible variables can be declared inside a Basis function (by prefacing their declaration by the additional reserved word `GLOBAL`). A global variable, that is, one declared outside of any function, is visible from any function. A local variable declared in a function is visible only within that function, where it hides a global variable of the same name.

CHAPTER

SEVEN

Built-in and Compiled Functions

There are three types of functions in Basis. The first, Basis language functions, are also called "user" functions. The other kinds are "built-in" and "compiled." Built-in functions are a special form of compiled function, either supplied as part of the Basis parser itself like `cos` or `iota` or `sqrt`, or written by a particularly Basis-skilled code developer. Compiled functions are ordinary Fortran routines whose calling sequence has been "taught" to the Basis interpreter.

Built-ins are usually used when it is desired to accept different kinds of Basis objects as arguments and return whatever type of object is appropriate. For example, many numerical-valued built-in functions will accept an arbitrary array of numbers and return an array of the same type, size and shape, whose entries were obtained by applying the function to each entry in the original array. Many of the most useful functions described below are designed to operate specifically on arrays.

For information on any individual function you can use `LIST` followed by the name of the particular function. In addition to giving you more information about what the function does, the output will also tell you what (if anything) the function returns, how many arguments it has, what their types are, etc. The Basis reference document also has more complete information. We discuss some of the more useful built-in and compiled functions in the following sections.

The *size* or *length* of an array is its total number of elements. The *shape* of an array is a vector whose components tell how many values the respective subscripts of the array can take on.

In Basis arithmetic, arrays must be of the same size and shape to participate in componentwise binary operations such as `+`, `-`, `*`, and `/`. The only exception is that one operand can be an array and the other a scalar, in which case the scalar is *broadcast*, which means that the operation is applied to the scalar versus every element of the array. Another way of thinking of it is that the scalar is expanded into an array of the same size and shape as the other operand, each element of which has the original scalar value.

Most functions which accept array arguments will also accept scalar arguments along with arrays, in which case they broadcast the scalars as described above.

Some of the functions below which accept array arguments don't care about shape, but only size. In this case they operate on corresponding components of their respective arguments, but you need to know what "corresponding components" are when the subscripts have different ranges. This is done in a fairly natural way: arrays in Basis, as Fortran, are stored in column major order (i.e., the first subscript varies most rapidly as we go through the array in memory). The elements of two arrays of different shape but the same size are said to *correspond* if they occupy the same relative position in this memory hierarchy.

When a function is called with an argument of the wrong type, or an expression involves mixed numerical types, Basis will perform automatic type coercion if necessary. For example,

```
3+4.
```

results in the “3” being converted to “3.0” before the addition operation, and if `f(x)` is a compiled function expecting a real argument `x`, then

```
f(3)
```

actually results in

```
f(3.0)
```

In an array of numerical constants of mixed types, its elements will be coerced to the highest type in the hierarchy integer $\rightarrow$ real$\rightarrow$ double$\rightarrow$ complex.

Likewise, most built-in functions try to do the right thing. For example, `sqrt(2)` means `sqrt(2.0)`.

Doubles are coerced to reals, and thence to integers, by truncation. Logical `true` converts to and from integer 1, and logical `false` converts to and from integer 0.

7.1 max and min Versus sup and inf

The functions `max` and `min` accept any number of arguments. If all arguments are scalar, then the result is the largest (resp. smallest) scalar in the list. If any argument is an array, then all other arguments must be either scalar or arrays with the same number of elements (but not necessarily the same shape). The scalar arguments, if any, will be expanded to vectors of this same length, with all entries equal to the scalar. Then the maximum (or minimum) will be taken component-by-component and returned as an array. The shape of this array will be the same as the shape of the first of the original arguments that was not a scalar.

In contrast, the functions `sup` and `inf` accept any number of arguments (even a single one), either scalar or array, of arbitrary size and shape, and return the scalar value of the largest (resp. smallest) component of all the arguments. Thus these functions always return a scalar; `max` and `min` will return an array if they have any argument which is an array.

A `max` or `min` of a single argument is treated as a `sup` or `inf`, since that is what you probably meant.

7.2 iota and spanl

The function `iota` is particularly handy if you wish to graph a function at an equally spaced set of points.

```
iota(n)
```

will give you a vector whose components are `1`, `2`, `3`, ... , `n`.

```
iota(m,n) #or iota(m:n)
```

will give you a vector whose components are `m`, `m1`+, `m2`+, ... , `n`. Thus, for example, you can get a vector of all the points a tenth of a unit apart in the unit interval `[0,1]`, and the corresponding values of a function `f`, by writing

```
real x = 0.1 * iota ( 0 , 10 ) , y = f ( x )
```

Note that in Basis you need not specify the dimension of a variable that is initialized with a vector or array when it is declared. It will be automatically dimensioned properly (with all subscripts based at 1). Note that `real x = 0.:1.:.1` would have accomplished the same result, as would `real x = 0.:1.:11`.

The function `spanl` is used to obtain a vector of points which are *logarithmically* spaced between two given points, rather than linearly spaced as one would obtain with `iota`. To get the eleven logarithmically spaced points in the interval `[0,1]` use

```
spanl ( 0., 1., 11 )
```

The first two arguments are the endpoints of the interval, and the third is the total number of points desired.

7.3 Information about Arrays: length, shape

Arrays are ubiquitous in Basis. Subscripting can be bizarre and shapes can change. These two functions allow you to obtain information about the size and shapes of arrays; the `shape` function also allows you to arrange the elements of an array into a different shape. This can be especially useful inside a function, where you may have been sent a perfectly arbitrary array as a parameter and you need to determine information about it.

7.3.1 The Function length

To find the size of (total number of elements in) an array, takes its `length`:

```
real a ( 3, 8 , 7 ) , b ( 5 , 5 )
function howbig ( x )
remark length ( x )
endf
call howbig ( a )
call howbig ( b )
```

will print out 168 and then 25. (`remark` is a Basis macro which simply prints out the value of its argument at the terminal.) Do not confuse `length` with `strlen`, which counts the number of characters in a character string.

7.3.2 The Function shape

The “shape” of an array is defined to be a vector containing the span of its subscripts as its components. The span of a subscript is the total number of values which it can assume, which if its upper bound is `hi` and its lower bound is `lo`, is given by `hi - lo` 1+. The function `shape` can do two distinct things for you; the first one is that if you send it a single argument, it will return you the shape vector for that argument. For example, with the above `x`, the value of `shape(x)` would be `[4,6,9,4]`.

The function `shape` also can be used to take a given array and change it to an array with a different shape. In this case we need to send it the shape vector of the result, as a second argument (or send the components of the shape vector as additional scalar arguments). For example,

```
shape (iota(64) , [4 , 4 , 4])
#or shape(iota(64),4,4,4)
```

will return a 4 by 4 by 4 array whose components in column major order are the numbers $1, 2, 3, ..., 64$. Why might one want to change the shape of an array? Well, one application that comes to mind is that you might want to add two arrays together componentwise, and they are the same length, so it ought to be possible. Unfortunately Basis will not allow you to perform binary operations on objects of different shapes. So you need to coerce one of the objects to the same shape as the other. For instance, suppose `a` is a 5 by 5 array and `b` is a 25 element vector, and we wish to add `b` to `a` componentwise, and leave the result in `a`. We could do this as follows:

```
a = a + shape ( b , 5 , 5 )
```

The shape of b is not permanently changed by this operation.

The shape function can also replicate an array to fill up a larger shape:

```
shape([1,2], 2, 3) = [[1,2],[1,2], [1,2]]
```

and

```
shape(1., shape(a))
```

is an array of 1.'s shaped like `a`.

7.4 Summing Arrays: sum

The function `sum` allows you to add up the elements of an array without having to write a loop to do it. It takes one or two arguments; in the one-argument case,

```
sum ( x )
```

all the elements of array `x` will be added, and the scalar sum returned. if `x` is a scalar, `x` will be returned.

In the two-argument case, the second argument specifies a subscript of the first; the result will be an array containing the sums of the elements of the original over all values of that subscript. Thus the result array will be one dimension smaller than the original, but the same shape in the other dimensions. For instance, if `x` has shape `[12,8,90,10]`, then

```
sum ( x , 3 )
```

will sum `x` over the third subscript and produce a result having shape `[12,8,10]`. That is, `sum(x, 3, y)(i, j, l)` is the sum of `x(i, j, k, l)` over all `k`.

Likewise, if `y` were a two-dimensional array (i.e., a matrix), then

```
sum ( y , 2 )
```

would produce a vector whose components were the sums of the corresponding rows of matrix `y`.

If you like `sum`, you might also like its cousin `psum` (partial sum) which is good for integrating things.

7.5 Vector Conditionals with where

```
where ( cond , x , y )
```

The first argument `cond` must be of logical type and the second and third, `x` and `y`, must be of numerical type. The length of `cond` must be matched by that of `x` and `y`, although one of them can be a scalar and the other an array. The array returned consists of an array the same length as `cond` with components equal to the corresponding component of `x` for those elements of `cond` which are true, and the corresponding element of `y` where `cond` is false.

`where` also has a two-argument form which returns just those elements of `x` for which `cond` is true (a "compress").

```
where ( a > b , a , b )
```

is equivalent to `max(a,b)` because `a > b` is an *array* of logicals of the same size and shape whose *components* are `true` or `false` according as the *corresponding components* of `a` are or are not greater than those of `b`. Thus `where` will now return an array whose components are the larger of the components of `a` and `b`. You could do the same thing with `max`; but the point here is that the logical condition could be a great deal more complex. In other words, `where` allows you to build much more general functions than `max` and `min`, although only on two arguments.

CHAPTER

EIGHT

Commands

8.1 The Basis Command Capability

Frequently when you are using Basis with a simulation code of some sort, the author will have written a number of commands which you will be using. In Basis, a command looks much like a command line in some operating system: the name of the command, followed by a list of arguments separated from one another somehow (usually by spaces or commas, but not always). A number of questions commonly arise with commands, in particular:

- What are the types of the arguments? (Specifically, some may be expressions to be computed, and others may be strings to be taken literally. Which are which?)
- What delimiters are allowed or required? (Clearly, if spaces are delimiters, then "`3 +4`" is two arguments, whereas if they are not, then it is one argument.)

Let's consider a little background first. Basis has a built in command capability that allows any function to be invoked by a command-line type of syntax. Consider, for example, a function defined as in a previous chapter:

```
FUNCTION w(namex)
INDIRECT y=namex
y(3) = 7.
ENDF
```

Using the regular Fortran-like Basis syntax, this function is invoked by the `call` statement:

```
call w("x")
```

Basis has a reserved word "`command`" which allows any function to be invoked by a command line syntax. In this example, `w` would be invoked by the statement

```
w command "x"
```

That is, we give the function name, the reserved word `command`, and then a list of the function's arguments. If there is more than one parameter on this list, the list may be delimited by either commas or spaces. (If this seems an arcane way to call a function, just remember that virtually all operating system commands have essentially this form: name of command followed by list of arguments.)

Using this command syntax, the arguments are all evaluated as expressions, and the default delimiters are spaces and/or commas. For example, in the command line

```
foo command "This is a string"  756 , , ( 912 + y )
```

the function `foo` is being called with four arguments:

- the character string `"This is a string"`
- the numerical value `756`
- a null argument (between the two commas)
- the expression `( 912 + y )`

The first two arguments are delimited by a space; the remaining ones are delimited by commas (spaces on either side of the commas do not count as delimiters when commas are present).

`NOTE:` Spaces outside parentheses act as argument separators; spaces inside do not.

It is important to emphasize that all of the arguments of `command` are taken to be expressions (in the above case, string-valued, numerical-valued, null-valued, and numerical-valued, respectively). Commas and spaces are taken to be delimiters (though not in combination—spaces around commas are ignored). Character string expressions must be quoted. If we had written

```
foo command This is a string  756 , , ( 912 + y )
```

then all of a sudden we would have seven arguments, and `This`, `is`, `a`, and `string` would be taken as identifiers to be evaluated.

The command capability allows the author to change these defaults. The author may specify different delimiters, either for the entire command or just between certain arguments; and can specify that some arguments be treated as if they were quoted strings, even if the quotes are not physically present. The details of this are covered in Chapter 10, "Deciphering Commands". This is done by suffixing an underscore "_" to "`command`" and then a sequence of letters specifying delimiters and argument types. Here is what we could do in the above case:

```
foo command_wSe This is a string , 756 , ,  ( 912 + y )
```

Immediately following the underscore is the lower case "`w`", which suppresses white space as a default delimiter. Thus only commas are valid delimiters in what follows. The upper case "`S`" specifies that the first argument (everything up to the first comma) is to be taken as a string, i.e., to be treated exactly as if it were quoted. The lower case "`e`" specifies that the second (and all remaining) arguments are to be expressions. Notice that since white space has been suppressed as a delimiter, parentheses are no longer necessary in the last expression.

What the author does to hide all of this from you is to define the command to be a macro which expands into the `foo command_wSe`. You can see what a command name really stands for by using the (you guessed it) `LIST` command.

In the chapter "Deciphering Commands" we go into more detail on this subject.

CHAPTER

NINE

Saving and Restoring Code and Data in Binary

9.1 The PFB Package

The PFB package can save and restore data, functions, and macros in binary form. The PFB package is not a required component of a Basis program; use '`list packages`' to see if it is present. (Note for when you start writing your own programs: PFB can be added to a program you make with Basis by adding the name pfb to the directory list input for `mmm`.)

Basically the process has three steps.

1. Create an output file:

   ```
   create myfile    # or whatever
   ```

2. Enter one or more `write` commands, which can take the following forms:

   ```
   write <namelist> # saves all the items named
   ```

   There are the following special forms of this command:

   ```
   write functions   # saves all user-defined Basis functions
   write macros      # saves all currently defined macros
   write variables   # saves all user-defined variables
   write all         # save functions, macros, and variables
   ```

3. When finished, close the file:`close`.

   The data is stored in a portable database format named PDB. The files can be moved to another computer and used there even if the new computer has a different data format.

9.2 Reading in Previously Saved Data

To read in all of the data you wrote do:

```
restore myfile            # reads all data from file myfile
```

To examine the data in the file without bringing it into your program permanently, you can use the `open` command:

```
real(8) x=3., y=4., z=5.
create myfile
write x,y,z
close
forget x,y,z  #x,y,z gone now
open myfile
x, y, z      #prints x,y,z in file
real(8) x=pfb.x #copy in just x
close
```

(You cannot assign to the variables in a file you have `opened`, you can only read the values).

See the manual page for PFB for fancier uses, such as comparing items from different files.

CHAPTER

TEN

Error Recovery and Diagnosis

10.1 Error Recovery

When an error occurs, it can be an error which Basis detects (such as trying to add a complex number and a character string) or one which is detected outside of Basis (such an floating point overflow). All errors detected by Basis result in a call to the routine `kaboom`. Whichever other errors a particular version of Basis can trap are trapped to a routine called `yuck` which in turn calls `kaboom`. This is why you may see a message '`yuck:  floating point error`' followed by messages about recovering to the prompt. Very rare but serious errors may cause an immediate program exit via `baderr`, and some system errors cannot be trapped, and exit without allowing Basis to regain control.

Assuming you reach `kaboom`, it either returns you to the prompt or causes the program to terminate. By default it returns you to the prompt. A routine `errortrp` is provided which can change this behaviour:

- `errortrp("on")` causes `kaboom` to recover to the prompt.
- `errortrp("off")` causes `kaboom` to terminate the program.

When an error occurs a trace file is written containing diagnostic information. To help you with a problem, the Basis staff needs to know what messages exactly were printed on the terminal when the error occurred, and what information is in the trace file.

To increase the information you get when an error occurs, we recommend setting

```
debug = yes
```

at the beginning of your session. If the trace files just annoy you no end, you can set `bastrace="none"` to eliminate them but this will make it hard for us to help you.

10.2 Syntactic and Semantic Errors

There are two kinds of errors that Basis can find.

- *Syntax* errors occur during the parsing of input code, and are caused by grammatically incorrect statements. Typical errors might be an illegal character in the input, a missing operator, two operators in a row, two statements on the same line with no intervening semicolon, unbalanced parentheses, a misplaced reserved word, etc.
- *Semantic* errors occur during the execution of the code, after it has been parsed as grammatically correct. These have to do not with how statements are constructed, but with what they mean. Such things as incorrect variable types or sizes, nonexistent variables, subscripts out of range, and the like, are semantic errors.

Basis is a single-pass parser, that is, it looks at its input only once. It also is a one-look ahead parser, meaning that at the most it is never looking more than one symbol ahead of the current context. By the time a syntax error has been detected, it is likely that a lot of the context information to the left of the error has already been lost. The diagnostic information that Basis gives attempts to be as useful as possible, but because of the very limited context information available, it is far from perfect.

Semantic errors are often possible to diagnose more precisely. We have attempted to make the semantic error information supplied as useful as possible. Sometimes some of the information is only useful to someone familiar with the internals of Basis; but we hope that in most cases it will help you find your error.

10.2.1 Syntax Errors

Here is an example of a statement containing a syntax error:

```
sum ( where ( a > v , ones ( length ( a ) ) , 0)
```

Let’s take a look at what Basis prints out as a result of this error:

```
sum ( where ( a > v , ones ( length ( a ) ) , 0 )
                                                ˆ Syntax error.
Attempting to parse after following context:
<lhs> ( <argitem>
which may not be followed by "cr" in this context.
Count of parentheses unbalanced: left = right +    1.
Expected one of the following (?):
  ) ,
Returned to user input level.
```

When the parser echoes the line being parsed, with “ `Syntax error`” underneath the line, the caret points to where the error was detected, not necessarily to where it occurred. In this case, the caret points past the end of the line, a clue that something is missing. The information about the parsing context is useful only to a Basis expert, but the statement that it can not be followed by `"cr"` (carriage return) is useful. That seems to say that the line is too short and reinforces our suspicion that something is missing. The next line points out that so far in the line there have been more left parentheses than right, and the next two lines confirm that maybe the parser expected a right parenthesis or a comma. The expression was missing a right parenthesis.

The list of expected symbols (as opposed to the one which actually occurred) is not 100% accurate. It may not contain all possible symbols which could occur in the given context; or worse yet, it could be such a long list as to be virtually unusable. In the above case it did contain the missing symbol, and it was not needlessly long. Below is a case where the list supplied by the parser is too extensive to be much help:

```
function f(x)
if ( x  > 0 ) then return 0
return 1
endf
```

The diagnostic produced by this error is:

```
     endf
     ˆ Syntax error.
Attempting to parse after following context:
function <funcdes> <eos> <stlist> if <ifexp> then <stlist>
which may not be followed by "endf" in this context.
Expected one of the following (?):
  ( + - : << >> ? Groupname [ ˆ ‘ break call chameleon character
complex complex-constant cr do double double-complex-constant
double-constant else elseif endif for forget function
hex-constant if indirect integer integer-constant list logical
name next octal-constant range read real real-constant return
string while whitespace \{ Returned to user input level.
```

What has happened here is a relatively common error—the programmer has not completed an `IF` statement. An `ENDIF` or `ELSE` clause has been omitted. Deeply buried in the list of “expected” symbols you will find these two reserved words, and also `ELSEIF`. It is possible to imagine a meaningful continuation of the program starting with any of the other symbols in the list, but the length of the list quite effectively hides the real clues in its depth. Unfortunately, a one-pass, no-backtracking parser with a one token lookahead can not apprehend the entire surrounding context as a human can; it only knows what symbols might, in some circumstances, lead to a correct statement if placed in the current position.

This example also hints at another problem with syntax errors: they may be discovered long after the actual error occurred. In this case, if an

`ENDIF` was intended prior to the `return 1` statement, the error was not detected until the `ENDF` was seen, after that statement had been consumed. There could equally well have been a hundred statements parsed before the `ENDF` caused the parser to detect the error. Thus our advice is that if you have trouble tracking down a syntax error, don't confine your search to the immediate neighborhood where it was detected. It could have been many lines previous.

10.2.2 Semantic Errors

Many times the Basis diagnostics for semantic errors make it very easy to discover what was wrong. For example, the statement

```
b = c + a
```

produces the following diagnostic (when `debug` is `yes`):

```
parasgn2: Shape mismatch between source and target in
assignment or append.
Right side (source) true dimension=  2 true shape=    10   10
Left  side (target) true dimension=  0 true shape=
parasgn: error in assignment to variable named ’b’.
Writing traceback info to file trace24589x
Returned to user input level.
```

Clearly the problem is that the right side is a 10 by 10 array, and the left side `b` is a scalar, so this is an illegal assignment.

Sometimes the traceback information can be useful; if you examine the traceback file (in this case `trace24589x`), you will find that it contains

```
Here is the information I have on where you were:
The error occurred in the assignment or append statement:
b = expression
The following lines contain clues(not facts) about the r. h. s.
c+a
Parser’s action number =   3(ASSIGN  ), program counter =    26.
```

Frequently a semantic error will be detected inside a function, or perhaps nested inside several function calls. The error printout may concern a variable or parameter local to the function where execution is taking place, and the name of the variable seems totally off the wall. For instance, consider the declaration and function call:

```
integer z ( 2 , 4 , 5 )
call f ( z )
```

This function call produced the following error diagnostic:

```
parfetch: trouble with object named ’barf’.
expression being subscripted has     4 subscripts but variable
only has 3 dimensions.
Writing traceback info to file trace24589x
Returned to user input level.
```

Where did the object named “barf” come from? Clearly it has three dimensions but four subscripts, but we can scarcely correct the error until we know where it was. In this case, the traceback file proves invaluable. It contains (in part):

```
Here is the information I have on where you were:
   A call to f containing
   A call to brf containing
   A call to arf containing
   the problem.
Error occurred in non-assignment statement.
The following lines contain clues to the error.
Some or all may be irrelevant to your problem.
1
barf
crf
crf
arf(crf)
arf(crf)
brf(x)
z
f(z)
Parser’s action number = 374(OUTPUT), program counter = 50.
Group: Locals_arf  Num Vars: 1
barf(2,4,5)
Number of dimensions is  3, lengths =      2     4     5
# # # some information skipped here
Group: Locals_brf  Num Vars: 1
crf(2,4,5)
Number of dimensions is  3, lengths =      2     4     5
# # # some information skipped here
Group: Locals_f  Num Vars: 1
x(2,4,5)
Number of dimensions is  3, lengths =      2     4     5
# # # some information skipped here
```

The function f, which we called, has called function brf, which called arf, where the error actually occurred. So the variable barf, which caused the trouble, is local to the innermost function arf, as we find out farther down the traceback. Finally, our variable z, and f’s variable `x`, and `brf`’s variable `crf`, and finally `arf`’s variable `barf`, all have the same dimensions. It must be that these are the names of these functions’ formal parameters; `z` has been passed down as a parameter all the way to a routine which expected a variable with four dimensions. Either we declared `z` wrong, or misunderstood what number of dimensions it was supposed to have, or else there is an error in `arf` which needs to be corrected.

Error diagnosis is (usually) a fairly straightforward problem, and we hope that these examples have helped illustrate how to diagnose and correct bugs. Our final advice is:

- Set `debug` to `yes` to get the maximum information.
- If the terminal output is not adequate to locate the bug, use the traceback file. (You can see its contents by typing

  ```
  !more tracefilename # name is given in diagnostic
  ```

  at the basis prompt.)
- Please be patient and actually read the error messages and trace files. We find many users do not do this. Errors, especially when you are busy, can generate strong emotions. If we knew how to generate one-line messages that exactly described every error, we would. We need to put out a lot of information to help people find difficult bugs; this means that often a lot of information is put out about a simple bug. We have tried to recognize this problem and put the more elaborate information in the trace file, so that it isn’t read except in more difficult cases.

CHAPTER

ELEVEN

Deciphering Commands

This chapter continues the discussion of commands. We do not recommend reading this chapter on a first reading of this manual. Or a second. Come here when you have a real problem with a command.

Commands defined by the author of a code are defined as macros. You need not know how to write a macro yourself in order to understand one somebody else wrote. A macro definition associates a name with a body of text, and this text is substituted for the macro name whenever it is encountered in the input stream. Macros may have arguments, in which case the arguments are expanded where they occur in the macro text. Thus a macro invocation can look just like a function call.

There are a few exceptions to macro expansion:

1. Text inside quoted strings is never expanded.
2. Macro expansion in an expression can be suppressed by enclosing the expression in braces { and }. (The braces are otherwise ignored by Basis.)
3. Macro expansion in a command argument can be suppressed by expressing its type with an upper case letter (in the preceding example, "`S`" as opposed to "`e`"). More about this later.

Enter this and see what happens:

```
list pi
pi
{pi}
```

the `LIST` command causes the macro definition of `pi` to be displayed. The second line will cause a funny display something like

```
3.14159265358979323 =    3.14159D+00
```

Left of the = sign is the text that was actually substituted for `pi`. Right of the = sign is the value of this, to the number of digits specified by the built-in variable `fuzz`. The third line causes an error because if we suppress the macro expansion of `pi` by enclosing it in braces, it then becomes an unknown symbol.

What does all this have to do with deciphering somebody's command? Well, suppose you are running somebody's simulation code with the Basis interface, and there is a certain command you want to use, and you are unsure what the arguments are supposed to be or how they are supposed to be delimited. Typically this individual will have defined this command as a macro, so the first thing you need to do is to track down the text of this macro. This is not hard to do; simply type in

```
list commandname
```

Basis has a command `timer` which is used as `timer on` and `timer off`. The text for `timer` is:

```
partime command_s
```

This means that `timer` calls a function named "`partime`." The "`command_s`" specifies (via the "`_s`") that it accepts at least one argument and that the argument will be an unquoted string with macro expansion enabled. "`s`" and "`S`" both express that an unquoted string argument is expected; upper case causes macro expansion in the argument to be suppressed. We don't know from the definition how many arguments the macro (or function) expects; but they will all be unquoted strings, if there are more than one—this is governed by the last letter in the specification (and here, the only letter, "`s`"). There are no delimiter specifications in this command, so white space and commas will be accepted. By the way, string arguments can be quoted, if you wish; they just don't have to be—*unless* they are to contain symbols that would be recognized as delimiters.

Argument specifiers can be `s` or `S` for strings (with and without macro expansion), and `e` or `E` for expressions (again, with and without macro expansion, but `E` is hardly ever used). You can specify a type for every argument by having a string of these characters, one per argument; but if, as is often the case, all the arguments from some point on are the same type, then Basis will keep using the last character in the string of specifiers. Thus `command_se` is the same as `command_seeee`... . Parentheses are used to specify repetition of more than one type, e. g., `command_e(Se)` is the same as `command_eSeSeSe`... .

Delimiter specifiers may be included in the specification string. If they are at the very beginning of the string then they determine the default delimiters for all arguments. If they occur between argument specifiers, they express what delimiter(s) would be valid just between those two arguments. Delimiter specifiers are `w/W` (suppress/enable white space), `c/C` (suppress/enable comma) `a/A` (suppress/enable at sign), and `q/Q` (suppress/enable equal sign). As has been previously mentioned, if no delimiters are specified then the default is "`WCaq`", i.e., white space and comma enabled, at sign and equal sign disabled.

Let us look at a few more examples from among the Basis predefined macros. Here is the expansion for `tek` in the `ezn` graphics package:

```
ezcdodev command_S(ScQS) tek $1
```

This calls a function named `ezcdodev`. The first argument is a string with no macro expansion (since this is the string “`tek`” itself, it is clear that we do not want to expand it in its own expansion). Subsequent arguments (if any) occur as pairs of strings with no macro expansion; the two arguments in a pair can be separated by equal signs or white space, but not commas (the “`cQ`” specification disables commas, enables equals, and does not change white space, which is enabled by default). Pairs are separated from other pairs by the default, white space or commas, because the “`cQ`” specification occurs only between the elements of a pair. The “`$1`” notation stands for the first argument of the macro call. (A macro may be defined with arguments, just like a function, in which case, when an invocation of the macro is expanded, `$1` will be replaced by the text of the first actual argument.)

Here is the macro text for `cgm`:

```
    ezcdodev command_SSc(SwcS) cgm $1
```

this calls the same function, but the arguments and delimiters are specified differently. All arguments are strings with no macro expansion. The first two are separated by white space or comma (the default), and the second is separated from the third by white space only (“`c`” suppresses comma). Subsequent arguments, it would appear, occur in pairs with white space or commas between the pairs, but the puzzling “`wc`” seems to say that the two strings of a pair have no delimiters between them at all! On first glance this seems to make no sense; but in fact, the effect of this is to concatenate the entire rest of the line into a single string and never find a fourth argument. Thus, in fact, this specification is really the same as

```
ezcdodev command_SScSwc  cgm $1
```

since no fourth argument will ever be collected; there is no delimiter possible to set it off from the third.

The following is the expansion of `plotm`:

```
ezcplotm command_(eWCQ)
```

All the arguments of this command will be expressions with macro expansion enabled, and the delimiters will be white space, commas, and equal symbols.

Finally, here is `resume`:

```
osresume command_es $1 $2
```

This calls the function `osresume` with the macro’s first two arguments (`$1` and `$2`), the first of which is an expression, and the second of which is a string, with macro expansion enabled in both. The delimiters are default.

INDEX