The Basis System, part 2

The Basis Development Team

November 13, 2007

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

COPYRIGHT NOTICE

DISCLAIMER

DOE Order 1360.4A Notice

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

Basis Input

Basis input can come from the terminal, from a file, or via recursive calls from within compiled code that is being executed. Statements are executed one at a time, immediately. However, statements which are part of a larger construct (such as a loop or IF test) are not executed until the entire construct is complete. When interactively entering such constructs, the prompt will change to give you a visual clue to the depth of the structure in which you are presently.

When an error occurs in a series of statements, you can assume that the statements before the one that caused the error have been executed; but if you make a mistake when entering a more complicated structure, you will need to begin again from the beginning. For example, if you are defining a function, and enter a statement that has improper syntax, the preceding part of the function is lost.

For this reason, it is usual to place complicated Basis input in a file and use the READ command to process it.

Basis-reserved words (like READ) are written in upper case throughout this manual for purposes of emphasis but they are also recognized by Basis if they are entered entirely in lower case.

CHAPTER

THREE

Basis Tokens

3.1 What Is A Token?

The tokens or terminal symbols of a language are the basic building blocks of that language. Tokens are the lexical entities from which statements in that language are constructed. They are analogous to the words in a spoken language. It will help in the discussion of the Basis Language to have an idea of what its tokens are before studying the language syntax. Tokens can be divided into the following categories:

- Alphanumeric These include identifiers and constants.
- Reserved words Identifiers that have a special meaning and may not be used in another way, such as the word IF.
- Non-alphanumeric These include punctuation, separator symbols, operators, and the like.

Comments and line-continuation symbols are not language tokens; they are delimited by symbols that have no significance in the Basis Language. The Basis-reserved words, built-in functions, non-alphanumeric, and alphanumeric tokens are described in later sections.

3.2 Special Characters

Some of the non-alphanumeric tokens have special interpretations in Basis:

- Blanks and spaces are significant in Basis. They act as token separators. The number of blanks (spaces) is irrelevant, however, as long as there is at least one. Thus, for instance, ELSEIF is one token; ELSE IF is two tokens.
- Semicolon and carriage return (the end-of-line character) may be used interchangeably by the user as statement separators.

- If \ (backslash) occurs as the last character on a line, the next end-of-line is ignored, and so allows continuation from one line to the next. This is the only way to continue a quoted string. A line is also continued if the last token on a line is a left parenthesis, a comma, or any logical or arithmetic operator such as `+, -, *, !, &,` etc.

- #acts as a comment delimiter, and causes the rest of the current line to be ignored except for the end-of-line. None of the special characters has any special meaning inside quoted strings or in comments.

3.3 Alphanumeric and ConstantTokens

3.3.1 Identifiers or Names

The names of variables must begin with a lowercase letter or a dollar sign. Subsequently, names can contain letters of either case, underscores, or digits. Names of variables must be 128 or fewer characters. Case is significant, so `joe` and `joE` are distinct variables.

A name can also be specified by enclosing it in single right quotation marks (apostrophes). In that case the name can include any characters except apostrophes, carriage-returns, or line-feeds. The enclosing apostrophes do not become part of the name; they simply allow names which do not obey the above rules to pass the interpreter.

Different variables can have the same name, if they appear in different packages in the search stack. (Please see “The Search Stack” on page 15 for a discussion of the search stack.) An identifier is taken to be the first one encountered in a package that has that variable name. To access a variable that is not in the top package or to distinguish between variables with the same name in different packages in the stack, add the package name as a prefix, and separate the package name and variable name with a period. For example:

```
pkg.name2
local.name3
global.name4
```

The variables in packages are organized into groups. A group name must begin with a capital letter, and again, can be prefixed with a package name followed by a period to identify it uniquely. *Global* and *local* are legitimate package names for user defined global and local variables.

Variables from packages attached to Basis are organized into groups. Group names can be up to 128 characters and can be abbreviated to any unique prefix. Any variables the user declares become members of a special group called *User*.

The identifiers `$`, `$a`, `$b`, `...`, `$z` are pre-declared. They have the chameleon property, which is discussed in “Declaring and Initializing Variables” on page 4. The variable `$` always holds the value of the last expression displayed.

3.3.2 Constants

An integer constant is a string of one or more digits, as in Fortran. A real constant in the form xx.xE+x must contain at least one digit, and either a decimal point or an exponent, or both. The exponent is expressed as e or E followed by an optional sign and at least one digit. Imaginary constants are either an integer or real constant followed by i or I. Spaces are *not* allowed between the number and the imaginary notation. Thus, 3I, 3.0i, 0.3E+1I, all represent the same imaginary constant. Double-precision constants are the same as real constants except that the letters d or D are used to denote the exponent.

String constants are delimited by double quotes (`"`). They must contain at least one character and can be composed of any printable ASCII characters; to include a double quote in a string constant, double it.

Two special forms of integer constants are also available: octal and hexadecimal constants. An octal constant is an octal number followed by b or B. A hexadecimal constant is a hexadecimal number, beginning with one of the digits 0-9, followed by an x or X.

Basis also contains the variables listed in “List of Parser Variables”, on page 32. These variables, such as “`pi`”, are available to the user for use in statements.

CHAPTER

FOUR

Declaring and Initializing Variables

The name of a user declared run-time variable must begin with a lower-case letter.

Users can declare run-time variables to be of type INTEGER, INTEGER(4), INTEGER(8), REAL, REAL(4), REAL(8), DOUBLE, LOGICAL, COMPLEX, COMPLEX(4), COMPLEX(8), CHARACTER, CHARACTER*(n), RANGE, INDIRECT, or CHAMELEON. The types CHAMELEON and INDIRECT are discussed in the following sections. Types REAL8 and COMPLEX8 (no parentheses) are also available, with the same meaning as REAL(8), COMPLEX(8), respectively.

Variables can be initialized in the declarations statement, as shown in the scalar declarations below:

```
INTEGER     x, y, z
INTEGER(4)  i4
INTEGER(8)  i8
REAL        i, j, k = 2.0
REAL(4)     x4
REAL(8)     x8
DOUBLE      d  = 2.d0
COMPLEX     c  = 2.0 + 3.0i
COMPLEX(4)  c4
COMPLEX(8)  c8
LOGICAL     l1 = true, l2 = false
CHARACTER*3 ch = "abc"
```

The variables x, y, and z are declared as integers of default size. I4 is an integer at least 32 bits (4 bytes) in length, and i8 is at least 64 bits (8 bytes) long. Variables i, j, and k are of type default real; k is initialized to 2.0. The variables x4 and c4 have at least 32 bits (4 bytes) precision independent of platform, and x8, c8 are at least 64 bits (8 bytes) in size.

Basis' use of *kind selectors* for integer, real, and complex data types is very similar to their use in Fortran 90. The discussion of precision above presumes that the underlying hardware is based on twos-complement integers and IEEE 754-standard floating point representations, in which case `real(4)` corresponds to IEEE single precision and `real(8)` to double. To restate this in Fortran 90 terms, a Basis `real(4)` kind should be the same as that resulting from `kind = selected_real_kind(6,38)`, and `real(8)` should match `kind = selected_real_kind(15,308)`.

Each individual variable to be initialized must be followed by an equal sign and value. To initialize `i`, `j`, and `k` to 1, 2, and 1, respectively, enter the following:

```
INTEGER i = 1, j = 2, k = 1
```

Variables which are not explicitly initialized are set to 0, or to blanks if they are of character type.

The variable called `autovar` controls whether or not declarations are required for all variables. See “autovar” on page 129.

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... 10, `y` from 1... 3 and 1... 5, and `z` from -3... 5 and 7... 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]
```

Vectors cannot be larger than the variables they initialize (except see the next paragraph), but they can be smaller, in which case only the first specified number of positions in the array will be filled. The initialization in a declaration follows the rules for assignment statements.

If an initial value is given, but no dimension is given on the variable being declared, the variable is created with the dimensions of the initial value. Thus the previous example could also be done this way:

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

Basis allows initialization expressions of arbitrary complexity, as long as all operands in them have values at run-time, since in fact such statements are ordinary assignment statements. References to functions are allowed as well. For example,

```
REAL a = sqrt(2) * ones(10,10)
```

defines a diagonal matrix `a` with the square root of two on its diagonal. For more details on expressions, see the next section in this manual, “Basis Expressions”.

The dimension specifications of declared variables are also allowed to be expressions of arbitrary complexity, as long as they are capable of evaluation at the time the declaration is executed. For example,

```
INTEGER i = 5, a(i,0:3*i-2) = 4, b(a(1,0))
```

declares and initializes `i` to 5, and then declares a to be an array subscripted 1...5 and 0...13, and then declares `b` to be subscripted 1...4.

As remarked previously, reserved words cannot be used as user identifiers. Previously declared variables or functions can be redeclared at any time, however. If the parser variable `debug` has been set to yes, Basis prints a warning message when a variable or function is redefined.

4.1 GLOBAL declarations

When a variable is declared it normally has global scope, that is, it will be known inside user-defined functions without any further declaration. If, however, a declaration occurs inside the definition of a user-defined function, the variable becomes local to that function invocation and will not be visible outside of it, and will vanish when the function returns. The user may override this by prefixing the keyword GLOBAL to any declaration inside a function, thus creating a variable which will be identical in scope to one declared outside of any function. For example,

```
FUNCTION phi(z)
  global REAL x = z/2.
ENDF
```

will create a variable `x` when function `phi` is called. Any existing global variable named `x` will be destroyed.

4.2 Package declarations

In addition to declaring global and local variables, the user may also declare a new variable to reside in an existing Basis package. When such a declaration is made, the variable is put in the last group of that package. Such a variable would be declared by a statement of the following format:

```
pkg type varname
```

where `pkg` is the name of the package in which to create the variable, `type` is the type of the variable (such as real or integer), and `varname` is the name or the variable.

EXAMPLE:

```
par REAL x = 3.1
```

The above example will create variable `x` in package `par`. The user can determine which packages exist in a given Basis code by typing,

```
LIST packages
```

4.3 Chameleon Variables

The variables `$` and `$a, $b, $c, ..., $z` exist when Basis starts. The variable `$` automatically assumes the value of the last expression evaluated in a display statement; the others must be explicitly assigned (but see the variable `autohist`, page 129.) When assigned a value, these

variables assume all of the attributes (e.g, type, size) of the value assigned to them. Hence, they are called chameleon variables. The user may declare other variables to have this chameleon property by using the type CHAMELEON, e.g.,

```
CHAMELEON abc = 3.45
```

causes `abc` to become a real whose value is 3.45. If a chameleon variable is currently an array then a subscripted assignment to the variable behaves like a normal assignment statement. Thus,

```
CHAMELEON abc = [1,2,3,4]
abc(3) = 5.6
```

results in `abc` being equal to `[1,2,5,4]` because the first assignment statement makes `abc` an integer array of length 4, and the second assignment statement has a subscript on `abc`, so its chameleon property is not invoked and the 5.6 is coerced to integer before being stored.

Except for `$` and `$a, ..., $z`, all variables that are assigned a value at execution time must exist, i.e., must have been declared. (The control variable `autovar` can be set to `yes` to change this). Formal parameters (the variables in the argument list) in user functions may not be declared.

4.4 Computed Names

It is possible to compute a name to be used in a declaration statement. This is done by surrounding a character expression with grave accent marks, as in this example which creates a variable named `x1` and initializes it to 3.0:

```
real  ‘ "x"//"1" ‘ = 3.0
```

4.5 Range Variables

A RANGE type is the same entity as the range used to subscript a variable. It consists of a low index, high index, and possibly an increment (negative increments are allowed), all separated by colons. An integer is also accepted as a range in which the low and high index are the same value.

EXAMPLE:

```
RANGE x = 3:5, y = 1:5:2, z = 5:2:-1, zz = 4
```

RANGE variables would eventually be used as subscripting information for an array. However, these variables can be passed in as arguments to a function and used within that function.Subscripting using a RANGE variable is identical to direct subscripting. Thus RANGE variables can have defaulted fields for their low index, high index, or increment (i.e. RANGE x = ::3).

The defaulted fields will take on the appropriate values for each array it subscripts. Some simple operations can also be performed on RANGE variables. You can add, subtract, or compare (i.e. ==, ¡>) two RANGE variables.

Three sets of examples and descriptions follow to illustrate

1. adding and subtracting RANGES, and the rules governing these operations
2. subscripting with RANGES and using DEFAULT fields
3. passing RANGES as arguments to functions

EXAMPLE OF RANGES WITH DEFAULT FIELDS:

```
RANGE a=2:10, b=: :3
integer z(a), y(6,7)
z(b)
y(4,b)
```

The above example will declare an integer vector `z` which is indexed from `z(2)` to `z(10)` and a 2D integer array dimensioned 6 x 7. The line "`z(b)`" will cause the values of `z(2)`, `z(5)`, and `z(8)` to be printed (just as if you entered `z(::3)`). The line `y(4,b)` will cause the values of `y(4,1)`, `y(4,4)`, and `y(4,7)` to be printed.

It should be noted that if you print the value of a RANGE variable which has a default low or high index, then any defaulted indices will be printed as a large negative number. Defaulted fields in a RANGE variable do not take on the "correct" value until it is used as an array subscript.

EXAMPLE OF ADDING AND SUBTRACTING RANGE VARIABLES:

The precise rules for addition and subtraction of ranges follow the examples.

```
range a=3:4, b=2:7:2, c=10:6:-1
a+4     ## results in 7:8,    remember 4 is the same as 4:4
c-4     ## results in 6:2:-1
b+b     ## results in 4:14:2
a+b     ## results in 5:11:2
b+c     #### illegal operation
```

When adding or subtracting ranges, the low indices of the operands are added or subtracted to produce the new low index and similarly the high indices are added or subtracted to produce the new high index. However, the resulting increment field is calculated in a different manner.

If the increments of both operands are 1, then the resulting increment is 1. If one operand has an increment of 1 and the other operand has an increment not equal to 1, then the resulting increment is set to the non-one value. If both operands have an increment other than 1, then these increment fields must both be the same value or else the operation is illegal. The resulting increment field is the same value as increments of both operands.

WARNING: Before adding or subtracting RANGES, you should always first call Basis function RNGSETDF to set any defaulted fields in the RANGE variable to the correct values. Adding or subtracting ranges with defaulted values which have not been reset by RNGSETDF will produce unexpected results.

EXAMPLE OF RANGE VARIABLES PASSED TO FUNCTIONS:

```
function density(x,y);  return mass(x,y)/volume(x,y); endf
function diffa(x)
  x=rngsetdf(x,2:10)   ## replace any default values of range x
  return a(x) - a(x-1)
endf
density(2:4, 1:10:2)
integer a(10) = iota(10)
diffa(3:7)
diffa(3: )           ## default value of high index in 3: is 10.
```

The calls to `density(2:4, 1:10:2)`, `diffa(3:7)`, and `diffa(3: )` will only calculate those values which are given in the ranged subscripts. In addition, the function `diffa` shows an example of range subtraction. This function makes a call to `rngsetdf` (a Basis built-in function) to replace any default values before doing the RANGE subtraction. Thus in the case when argument x is 3:, then x is reset to 3:10 before doing the subtraction. The function then returns the values a(3:10)-a(2:9).

4.6 The Colon Notation For Vectors

The notation `a:b:c` can be used with one or more real arguments to create linearly spaced arrays.

a:b:c with c real, a or b real

creates a vector containing values spaced at intervals spaced `c` apart. If `a`>`b`, the resulting vector will contain descending values. The vector created will be at least 2 long, and the first element will be `a` and the last will be `b`EXACTLY.

a:b:ic with ic an integer, a or b real

creates a vector of length `ic` of evenly spaced values from `a` to `b`. If `a`>`b` the resulting vector will contain descending values.

a:b with a or b real

defaults `ic` to the value contained in the control variable `ncolon`, whose default value is 100.

It is an error for `a` or `b` to be omitted if the other is real. It is an error for `c` or `ic` to be $<= 0$.

Note that the colon operator has a lower precedence than arithmetic operators, so to use a term `a:b:c` in an expression it will usually be necessary to enclose it in parentheses.

4.7 Indirect Variables

A variable declared to be type INDIRECT is actually an indirect reference to another variable. An INDIRECT declaration must include an initial value assignment setting the variable to the name of another variable, possibly including a package prefix, such as `"x"` or `"par.x"`. Any reference to an indirect variable after its declaration is equivalent to a reference to the variable named in the initial assignment. This assignment can only be changed with another INDIRECT declaration.

The variable which is being indirectly accessed may in turn be an indirect reference. INDIRECT can be used to write user functions which modify variables in their argument list; normal Basis functions pass arguments by value and such modifications do not

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

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

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

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

CHAPTER

FIVE

Expressions

5.1 Introduction

Expressions consist of operands, operators, and delimiters in a string specified by the grammar. Conceptually, we can consider operands as items that have value (e.g., constants, references to user variables that have a value, and invocations of functions that return values when executed). Operators are syntactic tokens that are usually described in terms of their semantic meanings, (i.e., what they are supposed to do at execution time). Unary operators produce a value from one operand, binary operators from two operands, and ternary operators from three operands. Finally, delimiters separate items (e.g., a comma-delimited list) and to change the semantic meaning of what they enclose (e.g., parentheses that change the precedence of enclosed operators).

5.2 Operands

String constants can be assigned to a variable, concatenated, built into arrays, passed to functions as arguments, etc. Everything that follows in this section addresses numerical and logical computations.

There are two types of expressions in Basis: expressions with numerical values denoted here by <exp>, and more complicated expressions, which, because they are allowed to have either numerical or logical values, are denoted by <lexp>.

The operands in <exp>s can be any of the following:

1. Integer, real, double, or imaginary constants.
2. Scalar variables of type integer, real, double, or complex.
3. References to arrays of type integer, real, double, or complex.
4. References to functions that return scalar or array values of type integer, real, double, logical, string or complex. There are three kinds of functions:

(a) Built-in functions are special functions that have been built into Basis. These are discussed in "Built-in Functions" on page 51.

(b) Compiled functions are Fortran functions that have been entered into a package database so that they can be invoked through the interpreter.

(c) User-defined functions are functions in the Basis Language defined by user commands, as explained later.

A reference to a scalar variable consists simply of its name if it is a user-defined variable, or if it refers to the top-most variable in the package stack by that name. Otherwise, it is referenced by a name of the form `pkg.name` where `pkg` is the name of the package in which it is defined.

A reference to a function consists of the name of the function followed by a list of expressions for its actual arguments in parentheses, as in Fortran or Pascal. Built-in and user functions are referenced in exactly the same way. If the function has 0 as an acceptable number of arguments, parenthesis are optional.

The operands in <`lexp`>s can be any of the operands allowed for <`exp`>s. In addition, <`lexp`>s can have operands of logical type, including the logical constants TRUE and FALSE. In some cases, logical quantities can also be organized and referenced as arrays. Array references and values of all types are discussed later in this chapter after a thorough consideration of scalar expressions.

5.3 Operators

The unary arithmetic operations are + and -. These two symbols also denote the binary operators "add" and "subtract". They have the lowest precedence of all operators. This means that in expressions containing other operators, add and subtract are evaluated last, as long as parentheses do not change the order of precedence. In expressions containing more than one of these operators, they associate to the left, which means that an expression such as

```
a + b - c + d
```

is evaluated as if it had been written

```
((a + b) - c) + d.
```

The binary operators "multiply" (`*`) and "divide" (`/`) have the next highest precedence, and are also left-associative. Thus, for example, in

```
a*b + c*d,
```

both the products `a*b` and `c*d` are computed, and then the addition is performed. In the expression

```
b/2*a
```

`b` will be divided by two, and then the result multiplied by `a`. To divide `b` by two times `a`, the expression must be written

```
b/(2*a) or b/2/a
```

The (scalar) arithmetic operator of highest precedence is the "exponentiate" (**) operator:

```
a**3
```

means a^3. Unlike the other arithmetic operators, ** associates to the right, so that

```
a**b**c
```

is evaluated as if it had been written

```
a**(b**c)
```

Operands of real, double, integer, and complex types can be intermingled at will in arithmetic expressions. In expressions containing a complex operand, the result is forced to complex type; if only integers and reals are present, the result is real. In expressions containing only integers, the result is always integer. In the case of division with a non-zero remainder, the quotient is taken to be the integer part of the result. For instance, 17/3 has the value 5.

The "matrix multiply" (`*!`) operator has its own peculiar size rules. The dot product operator (`!`), applies to objects of equal size. The dot product operator is included in the discussion of array operands later in this chapter. Thus, for the time being, we have considered all of the scalar arithmetic operators. We now discuss <`lexp`>s, those expressions which may produce logical values.

Operands for <`lexp`>s, those expressions that may produce logical values, can be built in three ways: from the logical constants TRUE and FALSE; from relational (i.e., comparison) operators between arithmetic values; and by combining previously computed logical values with the use of the logical operators. The binary relational operators are:

Operator	**Meaning**
= or == or .eq.	"equal"
<> or ~= or .ne.	"not equal"
< or .lt.	"less than"
<= or .le.	"less than or equal"
> or .gt.	"greater than"
>= or .ge.	"greater than or equal"

The equal and not equal operators can appear between operands of arbitrary type. If the types do not match, then coercion takes place in the order *integer* → *real* → *double* → *complex*.

The other four relationals are not meaningful for complex operands, so they can be used only with real, double or integer operands. Only the equals and not-equals operators can be used between character strings.

If the operands are not scalar a relational operator produces a logical array of the same shape as the operands;

```
(iota(5)=iota(5))
```

creates a logical array of length 5, all of whose elements = TRUE.

WARNING: The parentheses are essential here.

The relational operators are not associative, so two or more cannot be used in combinations like `a < b <= c`. Many languages allow such constructs syntactically, but they are almost always erroneous semantically. If `a`, `b`, and `c` are numeric, `a < b` is logical, and it is not legal to compare the logical `a < b` with the numeric `c`.

Listed in order of precedence, the logical operators are "not" (`~`) or (`.not.`), "and" (`&`) or (`.and.`), and "or" (`|`) or (`.or.`). The operands of `&` and `|` must be of logical type. Both `|` and `&` associate from the left.

Operators	**Precedence**
(subscripting, reference)	9 (highest)
**	8
* *! / ! // /!	7
+ −	6
:	5
= == ~= < > <= < >= >	4
~	3
&	2
|	1 (lowest)

5.4 Delimiters

The delimiters used in expressions are parentheses `(, )`, brackets `[, ]`, comma `,`, and colon `:`. We have given a few examples where parentheses were used to change (or emphasize) the order of operations. In order to understand the function of parentheses in expressions, consider first the rule for evaluating expressions without parentheses:

Evaluate operations in order of precedence, highest first. When there are multiple operations with the same precedence, evaluate the expression from left to right (except for **, which is evaluated from right to left).

When an expression contains parentheses, add the following rule:

Evaluate inside the most deeply nested set of parentheses first, then move outwards through the successive levels of nesting. Thus, parentheses can be thought of as operators that raise the precedence of operators enclosed within them to a higher value than those at any lesser nesting level.

For example, the expression

```
a*b + c*d
```

is perfectly legal, but both multiplies are performed before the addition. To force the addition to be evaluated first, rewrite the expression as

```
a*(b + c)*d.
```

As mentioned in the section on predefined and user-defined functions, parentheses are used to delimit the actual arguments of these functions. Thus,

```
sqrt(2.0*18)
```

returns 6.0, and

```
mod(17,5)
```

returns 2.

The actual arguments of a function can be any expression that evaluates to a meaningful type (one cannot extract the square root of a logical, for instance). Naturally, function references themselves can occur in actual arguments, as in

```
sqrt(sqrt(3 + mod(17,2))).
```

Finally, parentheses can be used to delimit subscript and subscript-range references for subscripted variables. Individual elements of an array are themselves scalars, and can be accessed by specifying a list of expressions in parentheses separated by commas. The number of subscript expressions specified must be less than or equal to the number of subscripts declared for the variable, and the values must be in the proper range. For instance, if we declare

```
INTEGER x(3:10), y(-5:1,6)
```

then `x(4)` and `y(1,1)` are legal references to elements of these arrays. A reference to `x(1)` is illegal because the subscript is out of range; `y(3,5,1)` is illegal because there are too many subscripts. If fewer subscripts are given than are declared for the variable, the unassigned elements (to the right) default to their minimum legal value. Subscript expressions, if not integer, are converted to integers upon evaluation.

In addition to references to single elements of an array, references to the entire array or to certain portions of it are allowed. This is discussed fully in the next section.

5.5 Array References and Operations

5.5.1 Subscript References

Any operand in an expression may be a reference to an entire array or to a non-scalar subset of it. In such a reference, the name of the array may be given alone, or followed by subscript specifications separated by commas. If subscripts are not present, then the entire array is taken to be the operand. When subscripts are present, the number of subscripts must be less than or equal to the dimensionality of the named array. Subscripts can be one of the following:

- Nothing The default low and high subscripts are used. These are the actual limits for that subscript.
- An integer Any expression that evaluates to a scalar. The scalar is converted to an integer if necessary. This subscript refers to a single entry.
- A range A range is specified by low:high or low:high:increment, where low, high, and increment (if present) are each any expressions that evaluate to scalars, or nothing. If low and/or high is omitted, the actual limit for a subscript is used. If increment is not present it defaults to 1. Zero is an illegal value for increment, but negative values for increment are legal. Expressions are converted to integer if necessary, and high must be greater than or equal to low (unless of course increment is < 0, in which case low must be greater than or equal to high).
- A vector of integers An arbitrary one-dimensional array of integers is allowed as a subscript of a one-dimensional array of numeric type. Naturally each element of the subscript array must be within the range of subscripts of the array being subscripted. If `x` is an array of variables and `i` is an array of subscripts, then `x (i)` is an array the same length as `i` whose entries are `x (i (1))`, `x (i (2))`, `x (i (3))`, ... `x (i)` can be a component of an expression or the object of an assignment. In the latter case, if there are repetitions in `i`, then the order of assignment is undefined.

The Basis Language has the unusual property that the user may subscript expressions, not just variable names. Subscripting has the highest possible precedence and multiple sets of subscripts are evaluated left to right. For example,

```
(x-y)(3:5)
```

is the vector `[x(3)-y(3),x(4)-y(4),x(5)-y(5)]`. In an expression, the lowest subscript of the expression is the common lowest subscript of the operands, if they agree, and 1 if they do not.

5.5.2 Dimensionality

Each array has a shape, expressed as a dimension n (0 to 7) and a string of n integers (i1, i2, ..., in) representing the length of the array in each dimension. When a variable is used in an expression,

the resulting object, after applying the subscripts, may have some of its dimensional lengths equal to 1. Each such component is dropped and the dimension of the object reduced accordingly. Thus, `x(5)` is a scalar (dimension = 0) and `y(3:7,6,2:5)` has dimension 2 and shape (5,4).

All operands in an array expression must be the same size and shape, or else be scalars. Basis automatically creates an object of the appropriate size and shape from any scalar in the expression. Thus, for instance

```
a(1:3,2:5) + 2
```

adds 2 to each element of an array whose first subscript is 1, 2, or 3 and whose second element is 2, 3, 4, or 5. On the other hand,

```
a(1:3,2:5) + b(1:3,2:4)
```

is illegal because the two sizes cannot be made to conform;

```
x(1:6) + a(1:2,1:3)
```

is illegal because the shapes (i.e., number of dimensions) are different.

When ordinary scalar operators, such as `*, /, +`, and `-`, are used among objects of the same size and shape, they represent component-by- operator. Thus,

```
a * b
```

multiplies the matrices `a` and `b` component-by-component. This is not matrix multiplication, for which there is a separate operator (See “Array Operators” on page 27.)

5.5.3 Subscripts on Basis-created Variables

Basis constructs variables for you in several cases:

1. An assignment is made to a non-subscripted chameleon variable.
2. An assignment is made to a non-subscripted variable, which doesn’t exist, and `autovar=yes`.
3. A function is called with arguments and the formal parameters must be created to contain the actual parameters.
4. A result is printed and `$` must be created to “remember” it.

In all of these cases, the new variable's lowest subscript in each dimension is the same as that of the item being assigned to it. The highest subscript is the lowest subscript minus one plus the length in that dimension.

Basis also creates temporary values during the computation of expressions. These have a lower subscript, a high subscript, and a stride that is used for labeling printed results. When an operation takes place, if all parties to the operation agree about things, the result continues to be of the same shape. (Scalars that are broadcast are treated as agreeing.) If the parties to the operation differ in strides, the result has stride 1. If they differ in lower subscripts, the result has lower subscript 1. These rules are applied on a per-dimension basis.

The built-in function `fromone` can be used to force lower subscripts and strides to 1.

5.5.4 The Square Bracket Operator

The square bracket operator can be used to build arrays. On the simplest level,

```
[3,4,5]
```

is a one-dimensional array whose contents are 3, 4, and 5. The following bracketed subscripts

```
[[1,2], [3,4], [5,6]]
```

represent a two-dimensional array whose contents are

```
1        3        5
2        4        6
```

Note that `[1,2]` is the first column, not the first row. (This was done for compatibility with Fortran, which stores arrays in column-major order). Expressions can appear in the array-builder brackets. For instance, given the declaration

```
INTEGER a(1:3, 1:3)
```

then the array expression

```
[a(,1), a(,2), a(,3)]
```

is exactly the same as `a`.

As another example, suppose that i is declared as follows:

```
INTEGER i = [22, 3, 45, 23, 2, 56]
```

Then if `x` is a one-dimensional array, `x (i)` is exactly the same as:

```
[x (22), x (3), x (45), x (23), x (2), x (56)]
```

If different arguments to the square bracket operator have different types, the result is formed by coercing all elements of an array to the same type in the usual hierarchy: *integer* → *real* → *complex*. Thus,

```
[1,2,5]   is integer,
[2,3,5.]  is real, and
[2,3,5i]  is complex.
```

The square bracket operator can also take a sequence of operands which are not all the same size and shape. The operator consumes its operands from left to right. At each stage, then, there are two operands, the result so far (call it s) and the next operand (t). First, s and t are coerced to the same type. Then, if s has zero length, the result is t. Otherwise, if t has zero length, then the result is s. Finally, assume that neither s nor t has zero length, and that ns and nt are the dimensions of s and t.

Let n = min(ns,nt), and let m be the largest dimension such that the size of s and t match in the first m dimensions. The result will have dimension m+1. The length of the m+1 direction will be the sum of the lengths of s and t considering them as arrays of dimension m+1 with the first m dimensions equal to their current value.

Thus, if s has shape (3,5,6) and t has shape (3,5,12) the result is of shape (3,5,18). If t has instead shape (3,4) then the result has shape (3,5*6+4) or (3,34). If t was a vector of length 3, the result would be of shape (3,5*6+1), since thinking of t as an array of dimension two its shape is (3,1).

This definition of the square bracket operator reduces to the correct result for the simple case when all the arguments to the operator are of the same size and shape. The square bracket operator always has a defined result as long as its arguments can be coerced to a common type.

5.5.5 Array Operators

The four array operators are "matrix multiply" (`*!`), "matrix divide" (`/!`), "transpose" (`transpose(x)`), and "dot product" (`!`) or (`.dot.`). (In a previous version of Basis, transpose was an operator; now it is a function; but we leave its description here for easy reference.)

The matrix multiply, matrix divide, and transpose operators are peculiar to arrays of two dimensions; they cannot be used in other contexts. Also, the matrix multiply operation `*!` must be distinguished from that performed by `*` written between two matrices, which simply multiplies the corresponding elements of the two matrices.

Two matrices multiplied with `*!` must have the property that the number of rows of the first equals the number of columns of the second (but the second can be one-dimensional and thought of as a column vector).

The result of the matrix divide operation

```
b /! a
```

is the solution x to the equation

```
a *! x = b
```

so that a*! (b /! a) is b. If a is singular, b /! a is an error. The numerator b may be a vector or a matrix; the result is of the same shape. If a is of type integer it is coerced to type real.

The transpose function transpose(x) exchanges the rows and columns of its operand. For example,

```
transpose(a(1:3, 2:7))
```

results in a matrix whose shape is (2:7, 1:3) and whose elements (i,j) contain the values that were in a (j,i).

The final array operator is !, the dot product operator, e.g.,

```
[1,2,3] ! [0,1,4] = 14.
```

The dot product can be applied between any two objects whose sizes are equal, regardless of shape. For example,

```
[[2,3],[4,5]]![1,2,3,4] = 2*1 + 3*2 + 4*3 +5*4 = 40.
```

Non-arithmetic operators can be used with array operands. &, |, and ~ can be applied to compatible arrays whose entries are logical values (i.e., true or false). The operations of = (==) and ~= (<>) can be applied between pairs of arrays of compatible size and shape whose elements are of any type. The remaining relational operators such as > can be applied between real and integer arrays.

In all these cases the result is a logical array. The built-in functions land and lor can be used to reduce logical arrays to the single logical value required in IF tests.

An important thing to emphasize again about size and shape is that any object with a single-value subscript range is an object of fewer dimensions. For example,

```
INTEGER x(1), y(1,5), z(5,1)
```

declares a vector x and matrices y and z; but when used in expressions, x is a scalar and y and z are vectors, not 1 x 5 or 5 x 1 matrices. Likewise the matrix product of a matrix and a vector is a vector, not an n x 1 matrix. Thus, if the declaration

```
INTEGER x(5,5), y(5)
```

is followed by

```
$a = x *! y
```

this implies that `$a` is a vector with 5 elements, not a 5 x 1 matrix.

5.6 The Concatenation Operator

The `//` operator has the same precedence as `*`, `*!`, `/`, `!`, and `/!`. It is called the concatenation operator and is defined in three cases: (1) both arguments equal to scalar character strings, (2) both arguments arrays or scalars of type logical, and (3) both arguments arrays or scalars of type(s) integer, real, double, or complex. The usual coercion rules apply in the latter case if the arguments have differing types.

5.6.1 Concatenating Character Strings

When its operands are scalar character strings, `//` simply performs string concatenation. For example, after:

```
$a="en"
$b="dow"
$c=$a//$b//"ment"
```

the variable `$c` will have the value “endowment”.

5.6.2 Concatenating Numerical and Logical Arguments

Two logical or numerical objects of any size or shape may be concatenated, producing a one-dimensional array whose total number of elements is the sum of the numbers of elements in the two concatenated objects, in the order in which they occur in memory (which means column- does things). For example:

```
$a=3//4
```

produces the vector `[3,4]`, while

```
$b=[[2,3],[4,5]]//[6,7]
```

results in `[2,3,4,5,6,7]`. If one wanted to produce a 2 by 3 matrix from this, which has `[6,7]` as its last column, one could use the shape operator, thus:

```
$b=shape($b,2,3)
```

forcing the concatenated result into the desired shape.

As a second example, consider the code fragment below. The routine "update" accepts the incoming values of the array `y` and the scalar `t` and returns new values. These new values are concatenated onto an accumulation of the older values, and then forced into a shape such that they can be displayed in rows with `t` in the first column and the corresponding `y`'s in columns 2, 3, and 4.

```
real y(3) = [1., 2., 3.], t = 0.
integer i
$a = t // y                        #initialize output
do i = 1, 10
     call update (&y, &t)          #note call by reference
     $a = $a // t // y             #add next solution
     t = t + 0.1
enddo
$a = transpose(shape($a, 4, 11))
```

Note the use of the transpose operator in the last expression. If this were not used, we would see t, then `y(1)`, `y(2)`, `y(3)`, etc., running down the columns if we printed `$a` out, instead of across the rows.

Logical arrays and scalars whose entries are logical values ("true", "false") may be concatenated following the rules above. Logical and numeric objects are incompatible and cannot be mixed in concatenations. Neither argument of a concatenation may be a structure, even if all of its entries are numeric. The result of such an operation, if it is attempted, will be unpredictable.

CHAPTER

SIX

Display and Assignment Statements

A display statement is simply a list of expressions separated by commas. When a display statement executes, the expressions are evaluated left to right, assigned to the special chameleon variable $, and then displayed. At the end of execution, $ has the value of the last expression computed For example,

```
3 + 1, 2
```

will display 4, then 2, and the variable $ will have the value 2 at this point. If a semantic error occurs during the execution of a display statement, execution of the remainder of the statement is aborted, and $ has the value of the last correct expression evaluated.

The variable `autohist`, 32.1, can be used to cycle the results through `$a`, `$b`,...,`$z` instead of always using $.

Note that only arithmetic and string-valued expressions, i.e., <`exp`>s, can be displayed in this way. The syntax of the display statement does not allow for the full generality of <`lexp`>s, which include logical-valued expressions. However, this limitation can be circumvented, and the values of logical expressions displayed, by placing parentheses around them, thus:

```
(x + y < 2)
```

Without this restriction we would be unable to translate the statement `a = b` because we could not tell if this is an assignment statement or a display of a logical expression.

The assignment statement has the general form

```
target = source,
```

where `source` is an <`lexp`> as described in the last section, (i.e., any expression, capable of evaluation, of any type, size, or shape). The target is the object where the value or values of the source object will be stored. If `target` is a scalar or a scalar element of an array, then the statement is a simple scalar assignment and needs no further explanation.

If `target` is a chameleon variable it assumes all the characteristics of `source` (size, shape, and type), and then receives the value(s) of the source object. It is not possible to generate an error

when an unsubscripted chameleon variable is the target object, unless the variable does not exist (i.e., has not been declared).

Array assignments are more difficult. Generally, if the target and the source expression are not of the same shape, it must be possible to store the expression as a subset of the target object. However, if the target is an array and the source is a scalar, then the scalar value will be broadcast to all specified elements of the array.

If the number of subscripts given in the assignment statement,

```
variable(subscripts) = expression
```

is less than the actual dimension of variable, it is assumed that the remaining subscripts have their lowest value (typically 1). If no subscripts are given in the assignment statement, the target is the full array variable. The shape of the target object may contain some 1's. The true shape of the target is its shape with the 1's dropped. There are two conditions on the true shape of the target:

- It must be of at least as many dimensions as expression.
- Each component of the target must be at least as large as the corresponding component of the expression.

If both these conditions hold, then `expression` can be stored as a sub-object of `variable`. Here are some examples:

If `x` has shape (3,2) then

```
x(2) = 5    sets x(2,1) to 5
x(2,) = [5,6]   sets the second row of x to [5,6]
x(2:,) = [[5,6],[7,8]]  sets the 2 by 2 submatrix of x whose upper
left corner is x(2,2) to the matrix

% MathFF:matrix[2,2,num[5.00000000,"5"],num[7.00000000,"7"],
% num[6.00000000,"6"],num[8.00000000,
% "8"]]

x(1:3:2,) = [[5,6],[7,8]]   sets the 2 by 2 submatrix of x
consisting of rows 1 and 3 and columns 1 and 2 to the matrix

% MathFF:matrix[2,2,num[5.00000000,"5"],num[7.00000000,"7"],
% num[6.00000000,"6"],num[8.00000000,
% "8"]]
```

The assignment `x(2,) = x(,2)` is erroneous: `x(,2)` has shape (3) while `x(2,)` has shape (2). If `x` had been a square two-dimensional array, however, this would have correctly set the second row to the second column.

If `y` has shape (5,6,7) then

```
y(3,2:6,1:7) = x
```

is a correct assignment. The target has shape (1,5,7). Its true shape is (5,7). The source has shape (3,2), which is smaller in each component. The assignment is performed beginning at `y(3,2,1) = x(1,1)`, `y(3,3,1) = x(2,1)`, `y(3,4,1) = x(3,1)`, `y(3,2,2) = x(1,2)`, etc.

Assignment is allowed to a one dimensional array subscripted by an arbitrary subscript array, e. g.

```
x ( [20, 13, 3, 56, 43, 5]) = y (3:9)
```

assigns `y (3)` to `x (20)`, `y (4)` to `x (13)`, `y (5)` to `x (3)`, etc. Note, however, that the result of an assignment to a variable with repeated subscripts, such as

```
x ( [20, 13, 3, 13, 43, 13]) = y (3:9)
```

is undefined. This is because on some architectures this assignment will be parallelized, in which case we do not know the order in which the assignments to `x (13)` will occur.

One special case requires some thought to understand. As an assignment target, `x` and `x()` are very different. In the latter case, one subscript has been given, although defaulted, and hence that one subscript defaults to its lowest possible value; and any other subscripts will then default to their lowest possible values, since they were omitted. Thus the target `x()` is the first element of `x`, while the target `x` is all of `x`.

6.1 Assignment Actions

For each variable, the user may specify a string containing Basis language statements called its assignment-action string. This string will be parsed and executed after each assignment statement in which the corresponding variable name appears on the left-hand side of the assignment statement. See 33.12.

6.2 Operator Assignments

The form of an operator assignment statement is

```
target op= source
```

where `op` can be any of the seven operators `+`, `-`, `*`, `/`, `|`, &, or `**`. Many readers are no doubt familiar with the operator assignments from C and C++. The above statement has the same effect as

```
target = target op source
```

and so it can be thought of as a shorthand notation. For example,

```
i = i + 1
```

can be written with fewer keystrokes as

```
i += 1
```

This is especially handy if the left-hand side of the assignment is a long identifier with many subscripts.

The left side of an operator assignment can be a one dimensional array with an arbitrary array of integer subscripts. However, if any of the subscripts is repeated, then the resulting element with that subscript is not defined.

6.3 The Append Statement

The append statement is part of a facility in Basis which assists in the process of collecting lists of values, such as time histories, in an efficient manner. The components of the facility are:

- `setlast`, a routine for imposing a limit on the last subscript.
- The `:=` "append" operator.
- `rtadddim`, a routine for "adding" a dimension to a variable.

The routine `setlast(`*`name, n`*`)` limits the LAST dimension (only) of the variable name to length of *n*. If *n* is greater than the current length (unlimited) of last subscript of name, then an attempt is made to expand storage so that the length will be at least *n*.

If *n* is greater than the current maximum value, then the maximum is set to 1.5 times the existing value or at least 16. This exponential growth is used to help reduce memory fragmentation while preserving constant time operation cost. `setlast` can be used on static arrays as long as no attempt is made to exceed the actual storage available.

The append operator `:=` works as follows: `x := y` is equivalent to storing `y` after the current end of `x`, increasing the final subscript of `x` appropriately (using `setlast`'s internal routine `rtsetdl`). `y` must be of an appropriate shape to be so stored. If `y` is of the same dimension as `x`, `y` is viewed as an array of values to be added, and the final subscript of `x` will increase appropriately. If `x` is a scalar, it is first made a one-dimensional vector of length 1.

`rtadddim`("name") adds a new subscript of 1 to name. This can be useful in setting name up as a target for a `:=`.

Example 1:

```
real x(3,3)
call setlast("x", 2)  # x will act as if it is shaped (3,2)
```

Example 2:

```
real x(3,3)
call setlast("x", 0)  #x will act as if it is shaped (3,0)
x:=iota(3)   # now x is (3,1) (but storage is still (3,3) )
x:=iota(3)   # now x is (3,2) (but storage is still (3,3) )
x:=iota(3)   # now x is (3,3) (but storage is still (3,3) )
x:=iota(3)   # now x is (3,4) (but storage is now   (3,16) )
x:=iota(3)   # now x is (3,5) (but storage is still (3,16) )
x:=[iota(3),iota(3)]
             # now x is (3,7) (but storage is still (3,16) )
```

Example 3:

```
integer y(0) # set up an empty array
integer i
do i=1, 1000
   y:=i
enddo
# After this loop, y is the same as iota(1000)
```

6.4 The Logical IF Statement

The IF statement in Basis takes two forms that are similar to the Fortran logical IF and block IF statements. We use this same Fortran terminology when referring to the two IF statements in Basis.

The syntax for the logical IF is

```
IF (<lexp>) <nonnullstatement>
```

where semantically <lexp> must evaluate to a scalar logical value. The <nonnullstatement> can, but need not, be on the same line as the IF (<lexp>). Furthermore, unlike Fortran, the only restriction on the type of controlled statement is that it cannot be null. Thus, in principle, logical IFs (and other structured statements) can be nested to any depth.

In the following example,

```
IF (a < b & b < c) m = c
```

sets `m` precisely to `c` if both `a < b` and `b < c` are true. A more complicated example is

```
IF (i <= maxindex)
     IF (a(i) ~= 0)
          b(i) = b(i) / a(i)
```

The nested logical IFs illustrated above are not equivalent to the single IF statement

```
IF(i<=maxindex & a(i)~=0) b(i) = b(i)/a(i)
```

because in the evaluation of a conjunction (expression with `&`), both operands of the conjunction are always evaluated even if the first operand is false. Thus, if `i > maxindex`, an attempt would still be made to evaluate `a(i)`, which would cause a semantic error at run-time (subscript out of range). For this reason, the first form is preferred.

Like all Basis statements, IF statements are actually compiled into a low-level code, and this code is not interpreted (i.e., executed) until the complete statement has been read in. If errors in syntax (i.e., the grammatical form of the statement) are detected during the compilation process, compilation is aborted and the offending statement must be retyped. Once a statement is entered correctly, it executes to completion unless the detection of a semantic error aborts execution. If execution was nested inside one or more structured statements, user functions, or both, when the error occurred, information about the nesting is displayed.

The normal Basis prompt at initialization is[1]:

```
Basis>
```

During the input of structured statements, however, this prompt changes to a series of > symbols that indicate the nesting level. The prompt returns to normal after execution completes. For instance, the example above with Basis prompts is:

```
Basis > IF (i <= maxindex)
\>              IF (a(i) ~= 0)
>>                   b(i) = b(i)/a(i)
Basis >
```

6.5 The Structured IF Statement

The other type of IF statement is quite similar to the Fortran block IF. In skeletal form, it looks like this:

[1] In an application code the main prompt is usually changed by the author

```
IF(<lexp>) THEN
   <stlist>
ELSEIF (<lexp>) THEN
   <stlist>
...
     ELSE
   <stlist>
ENDIF
```

where <stlist> represents either a single statement or a sequence of statements separated by semicolons or carriage returns.

ELSEIF and ENDIF must be single words. If you enter `ELSE IF`, for instance, then the compiler will think that a new IF statement, nested inside the current one, is being started. `END IF` will cause compilation to abort with a syntax error. The ELSEIF clause is optional. Also note that the ellipsis above indicates that there may be many ELSEIF clauses. The ELSE clause is optional also, but, of course, there can be no more than one ELSE clause.

Basis is not overly particular about the placement of THEN; it can be on a separate line, and it can, but need not, be followed by a statement on the same line. In fact, THEN can be omitted from an ELSEIF clause, provided that <stlist> begins with a non-null statement. Thus, although THEN is not syntactically important, ENDIF, ELSEIF, and ELSE are. ENDIF, ELSE, and ELSEIF must each appear at the beginning of a separate line, or be separated from <stlist> by a semicolon.

Here is a block IF that determines the maximum of two numbers:

```
IF (a>b) THEN
     m = a
ELSE
     m = b
ENDIF
```

This could equally well be written

```
IF (a>b)
THEN m = a
ELSE m = b
ENDIF
```

or could even be written on one line as

```
IF (a>b) THEN m = a; ELSE m = b; ENDIF
```

Semicolons are required to separate statements that appear on a single line.

The following nested block IFs determine the maximum of three numbers:

```
IF ( a>b ) THEN
     IF (c>a) THEN
          m = c
     ELSE
          m = a
     ENDIF
ELSEIF (c>b) THEN
     m = c
ELSE
     m = b
ENDIF
```

Of course, the built-in function max(a,b,c) is a lot easier!

CHAPTER

SEVEN

WHILE Statement

7.1 WHILE Statement

The WHILE statement is a loopingconstruct similar to that found in C:

```
WHILE (<lexp>)
     <stlist>
ENDWHILE
```

As in the IF statement, <lexp> is required to evaluate to a logical value (true, false). Otherwise a semantic error occurs and execution is terminated. At execution time, <lexp> is evaluated. If it is false, execution of the WHILE loop is terminated. If the WHILE loop is nested inside another statement, then control goes to whatever follows the ENDWHILE. If <lexp> is true, then <stlist> is executed. If <stlist> does not contain a NEXT, BREAK, or RETURN statement in its flow of control, then <lexp> is evaluated again and execution proceeds as above. (The NEXT, BREAK, and RETURN statements alter the flow of control and may cause the loop to terminate.)

For example, the following sequence of statements adds up the positive elements in array a. Note the IF statement nested within the WHILE.

```
sumpos = 0
i = 1
WHILE (i<=amax)
     IF (a(i)>0) sumpos = sumpos + a(i)
     i = i + 1
ENDWHILE
```

Below is a set of nested loops that compute the product of two square matrices a and b and place the result in c:

```
i = 1
WHILE (i<=n)
```

```
     j = 1
     WHILE (j<=n)
          c(i,j) = 0
          k = 1
          WHILE (k<=n)
               c(i,j) =  c(i,j) + a(i,k)*b(k,j)
               k = k + 1
          ENDWHILE
          j = j + 1
     ENDWHILE
     i = i + 1
ENDWHILE
```

The Basis prompt on the innermost loop will be >>>.

7.2 BREAK and NEXT Statements

The BREAK statement provides a way to exit from a loop other than via the controlling condition going false. Indeed, a WHILE (true) will never terminate unless it contains a BREAK. For example, the following example is equivalent to the first WHILE in the first example given above:

```
sumpos = 0
i = 1
WHILE (true)
     IF (i<=amax) THEN
          IF(a(i)>0)sumpos = sumpos + a(i)
          i = i + 1
     ELSE
          BREAK
     ENDIF
ENDWHILE
```

Normally, as is the case here, the BREAK statement will be controlled by some sort of test.

BREAK can be used to exit from nested structures by using the form BREAK n (or BREAK(n)), where n is the level of loop nesting. BREAK and BREAK 1 mean the same thing.

Here is an example using BREAK from a nesting level of three deep:

```
i = 1
WHILE(true); j = 1
     WHILE (j<=5); k = 1
          WHILE(k<=5)
               IF(i*j*k > 86) BREAK 3
```

```
                k = k + 1
            ENDWHILE
            j = j + 1
      ENDWHILE
      i = i + 1
ENDWHILE
```

The nesting level expressed in a BREAK can, but need not, be enclosed in parentheses. It must be a positive integer constant, however, not an expression or variable name.

The BREAK statement is not used exclusively in WHILE statements; it can be used inside any iterative statement (iterative statements are discussed later). Bear in mind that a specified nesting level is the level of nesting inside loops only; the fact that each BREAK in the examples above is in an IF does not affect its level as far as loops are concerned. A BREAK statement that occurs outside a loop, or one that occurs inside a loop with an expressed nesting level greater than the actual nesting level, will simply be ignored and has no effect whatsoever.

The NEXT statement has a provision for prematurely reentering a loop (including an outer loop that contains the loop with the NEXT statement). For example, the following loop adds all the elements of array `a`, except those whose subscript is divisible by 5:

```
i = 0; sum = 0
WHILE (i<n)
      i = i + 1
      IF(i/5*5 = i) NEXT
      sum = sum + a(i)
ENDWHILE
```

The integer `i/5*5` is equal to `i` precisely if `i` is divisible by 5, in which case NEXT causes control to return to the top of the loop, where `i < n` is checked again.

In the NEXT statement, as in BREAK, an optional nesting level can be given, either as an absolute integer or as an integer in parentheses. NEXT and NEXT 1 are equivalent. NEXT 2 causes iteration to proceed to the top of the next outer loop, and so on. As with BREAK, a NEXT is ignored if its expressed nesting level is greater than the current actual level, or if it occurs outside a loop.

CHAPTER

EIGHT

FOR Statement

The FOR statement, except for minor syntactic variations, is similar to the one in the C Language; C programmers should beware the interchanged roles of comma and semicolon.

Its general form is

```
FOR (<forinit>, <lexp>, <stlist2>)
     <stlist1>
ENDFOR
```

where <forinit> is a (possibly null) list of one or more assignment statements, separated by semicolons or carriage returns. These initializations are performed exactly once, when the loop is first entered from above. The logical expression <lexp>controls iteration. If it evaluates to false, the loop is exited; and if it is true, <stlist1> executes, then <stlist2>, and then <lexp> is tested again, etc. BREAK works exactly as described in the preceding section, while NEXT transfers control to the execution of <stlist2>.

Logically, the FOR loop above is equivalent to

```
<forinit>
WHILE (<lexp>)
     <stlist1>
     <stlist2>
ENDWHILE
```

except that NEXT transfers control to <stlist2>.

Normally <forinit> might be used to initialize a loop control variable, <lexp> to test it, and <stlist2> to increment it at the end of the loop. For example

```
sum = 0
FOR(i = 1, i<=n, i = i + 1)
     sum = sum + a(i)
ENDFOR
```

adds up the elements of array `a`. This could also be written as

```
FOR (i = 1; sum = 0, i<=n, sum = sum + a(i); i = i + 1)
ENDFOR
```

or even

```
FOR (i = 1
     sum = 0,
     i<=n,
     sum = sum + a(i)
     i = i + 1)
ENDFOR
```

The semicolons separating statements are not required if statements are on separate lines. However, the commas between the three parts of the FOR header are required.

Note that ENDFOR must be preceded by a semicolon or carriage return.

Here is a matrix multiply using a FOR:

```
FOR (i = 1, i<=n, i = i + 1)
     FOR (j = 1, j<=n, j = j + 1)
          a(i,j) = 0
          FOR (k = 1, k<=n, k = k + 1)
               c(i,j) = c(i,j) + a(i,k)*b(k,j)
          ENDFOR
     ENDFOR

ENDFOR
```

CHAPTER

NINE

DO Statement

9.1 Uncontrolled DO

There are three forms of the DO statement. The first, called for obvious reasons the uncontrolled DO, is the simplest in form:

```
DO
     <stlist>
ENDDO
```

The initial DO must be followed by, and the ENDDO preceded by, a semicolon or carriage return, as indicated above. In this uncontrolled DO, <stlist> repeatedly executes; in fact, if it does not contain a BREAK statement, or if it does and it never executes, then the loop repeats forever.

9.2 DO-UNTIL

The second type of DO, DO-UNTIL always executes its body once, and performs the test at the end.

```
DO
     <stlist>
UNTIL (<lexp>)
```

In the DO-UNTIL loop, <stlist> executes and then <lexp> is tested. If <lexp> is true, the loop terminates; if it is false, <stlist> repeats, and so on. The logical expression must evaluate to a logical scalar at run-time, or a semantic error will occur.

BREAK works exactly as it does in other types of loop to effect exit. NEXT works in a reasonable way, but maybe not as one might expect without a little thought. NEXT causes control to proceed directly to the top of <stlist>, bypassing the UNTIL (<lexp>, on the philosophy that when reinitiated, this type of loop always executes its body once before testing at the end.

9.3 Controlled DO

The third type of DO is appropriately called the controlled DO, because its iterations and termination are controlled by an explicitly named scalar whose initial and termination values (and optionally, increment) are specified prior to execution of the loop. This construct is quite similar to the one from Fortran:

```
DO <lhs> = <init>, <term>, <incr>
     <stlist>
ENDDO
```

The controlling scalar <lhs> must be integer and scalar; unlike FORTRAN, it may be an element of an array. The other loop specifications, <init>, <term>, and <incr> must be scalar expressions with numeric values which can be coerced to integer. The ", <incr>" can be omitted. If it is, it defaults to 1 as in Fortran.

The controlled DO is roughly equivalent to the following statements, where %C1 and %C2 can be thought of as variables accessible only to the Basis run-time system that cannot be changed by the user:

```
     <id> = <init>
     %C1  = <term>
     %C2  = <incr>           # Or 1, if <incr> is absent
DO
     IF (%C2 > 0 & <id> > %C1) BREAK
     IF (%C2 < 0 & <id> < %C1) BREAK
     <stlist>
     <id> = <id> + %C2
ENDDO
```

Thus, the control expressions <term> and <incr>are evaluated exactly once, when the loop is entered. This is important because it enforces the concept that the loop will execute a number of times that is known upon entry, and that the number of iterations will not change subsequently, even if the user alters components of the expressions <term> and <incr>. Likewise, if the loop controlling scalar is a subscripted variable, its subscripts are evaluated exactly once, before the loop is entered. Even should these subscripts change within the loop, the same array element will still be used for the loop control. Finally, note that the test for loop exit is at the top of the loop, and that the incrementing is at the bottom (after each execution of the body).

The next DO loop squeezes the zeroes out of an array:

```
j = 1
DO i = 1, maxa
     IF(a(i) = 0) THEN
          maxa = maxa-1
```

```
     ELSE
          a(j) = a(i)
          j = j + 1
     ENDIF
ENDDO
```

The loop executes as many times as there were elements in a at the time the loop was entered, because the initial value of `maxa` is saved and used for loop control. Upon exit from the loop, `maxa` will have been changed to reflect the size of the smaller array.

The following example is, again, the matrix multiply; this time it is performed using several different DO loops:

```
i = 1
DO
     IF (i>n) BREAK
     DO j = n, 1, -1     # note negative increment
          c(i,j) = 0
          k = 1
          DO
               c(i,j) = c(i,j) + a(i,k)*b(k,j)
               k = k + 1
          UNTIL (k>n)
     ENDDO
     i = i + 1
ENDDO
```

CHAPTER

TEN

Functions Listed by Type

Basis contains three different kinds of functions: user-defined, built-in, and compiled. The latter two must be distinguished because there are different rules for using built-in and compiled functions. The built-in routines are documented in the following section. The compiled functions are documented in Chapter 33. The following tables are intended for browsing to locate the routine you need. They list the built-in and compiled functions classified by their general function or nature. Compiled functions are listed in *italic* face.

10.1 Common Mathematical

abs	aint	anint	exp	log	log10
alog	alog10	nint	ranf	sign	sqrt
mod	min	max	sup	inf	

10.2 Trigonometry

acos	asin	atan	atan2	cos	cosh
cot	sin	sinh	tan	tanh	

10.3 Type Conversion and Complex Numbers

aimag	cmplx	conjg	dble	float	int
sngl	struct				

10.4 Arrays

Most of the built-in functions can take arrays as arguments or produce them as output. These functions are helpful in working with arrays:

ave	cumaddin	diag	iota	land	lor
length	load	max	min	mnx	mxx
ones	outer	psum	ptp	ranf	rangex
rsum	shape	struct	sum	sup	inf
setlimit	*setshape*	where	gather	fromone	trueshape
truerange	setact	spanl	rmsdv	squeeze	setlast
rtadddim	*sorti*	trans-pose			

10.5 Character Manipulation

len_trim	index	trim	triml	trimr	substr
format	toupper	tolower			

10.6 Special Purpose

format	index	load	range	shape	struct
type	*help*	*news*	*dec*	*oct*	*hex*
allot	change	*basfree*	*gallot*	*gchange*	*gfree*
execuser	*comment*	*exists*	*flushlog*	*swset*	*switch*
protect	*paws*	*setmnarg*	*kaboom*	*parsestr*	setranf
getranf	seedranf	mixranf	cd,chdir	setenv	getenv
disk-	space				

10.7 Obtain/Set Scalar Values

ibasis	*rbasis*	*dbasis*	*cbasis*	*lbasis*	*sbasis*
sibasis	*srbasis*	*sdbasis*	*scbasis*	*slbasis*	*ssbasis*

CHAPTER

ELEVEN

Built-in Functions

The user has access to a number of built-in functions, which are invoked in an expression by using the name of the function followed by a parenthesized list of its actual arguments. Basis can return not only scalars, but also array values or even what are called structures. A structure is an array whose individual elements can be objects of different types: scalars, arrays, or even other structures.

In this section, there is a brief alphabetic list of currently implemented built-in functions, their required parameters, and a description of what they return. Generally speaking, the built-in functions allow the user license in what arguments can be sent. The number of arguments is checked, but frequently the type of argument can be virtually anything, and a correct result will be returned. Most of the arithmetic functions, for instance, will accept arguments of any size and shape, apply the function to each component, and return an object of the same size and shape with the new components.

In what follows, unless otherwise noted, a single argument can be a scalar of type integer, real, double, or complex, or an array whose elements are of these types. A function is applied component by component to arrays. Unless otherwise noted, the result components are of the same type, unless they need to be coerced to real or complex.

This list can be obtained at run-time with the command "`list Builtin`".

abs(x) returns the absolute value of object x.

acos(x) returns the inverse cosine (in radians) of object x. The inverse trig functions do not return complex values (e.g., acos(2) = 0). If x is complex (or has complex components), acos is applied only to the real part(s) of x.

aimag(x) returns the imaginary part of object x. Use float to get the real part of complex x.

aint(x) returns the integer part of object x (i.e., truncates the fraction.) The result is always real. If x is complex, aint is applied to the real part of x.

alog(x) alog(x) = natural logarithm of x

alog10(x) alog10(x) = base 10 logarithm of x

anint(x) returns the nearest (real) whole number to x. See "aint(x)".

asin(x) is the inverse sin (in radians) of x. See “acos(x)”. 11

atan(x) is the inverse tangent (in radians) of x. See “acos(x)”. 11

atan2(y,x) is the inverse tangent (in radians) of the angle between the positive x axis and the vector whose components are (x,y). x and y can be vectors of the same length, or both scalars, or either can be a scalar and the other a vector. See “acos(x)”. 11

ave(x,idim) returns the average of array x. x can be of type complex, real, double or integer. The resulting type is the same as x, unless x is of type integer. In this case, the resulting type is real. If idim is supplied, then the function is applied to each group of elements in x whose indices vary only in the i-th dimension. The output array produced is the same shape as x, except that the i-th dimension is removed. If idim is not supplied, then the function is applied to the entire array, resulting in a scalar output.

cmplx(x), cmplx(x,y) returns x if x is complex; if x is integer or real, it returns a complex number with imaginary part = 0 and real part = x (componentwise if necessary). In the two argument form, returns the complex number (x,y). It is a semantic error if x and y are not real or integer, or if x and y are not the same shape (except that one could be a scalar, which would be broadcast).

conjg(x) returns the complex conjugate of object x.

cos(x) returns the cosine of object x, which must be in radians. x can be integer, real, double, or complex; cos(x) will be real, double or complex as necessary.

cosh(x) returns the hyperbolic cosine of object x. See “cos(x)”.

cot(x) returns the cotangent of object x. See “cos(x)”.

cross(x,y) returns the cross-product of two real 3-d vectors x,y.

cumaddin(&x(i),y `x` and `y` are one-dimensional arrays of numeric type, and `i` is an arbitrary integer array of subscripts into `x`. `i` and `y` are of the same length. Note that the ampersand on `x` is required, because the contents of `x` will be changed by this operation. The effect of this function is the same as the Basis loop do j = 1, shape (y) x (i (j)) += y (j) enddobut of course it is faster because the operations are done by compiled code. Note that if `i` contains repeated subscripts, then the effect of this is to accumulate the sum of corresponding values from `y` into the `x` values corresponding to the repeated subscripts. Also note that if `i` does not have repeated subscripts, then it is much easier to do this as x (i) += yThe latter will not work, however, if `i` has repeated values, because only one of the sums will be assigned, and furthermore, it is impossible to know which, if the computation is parallelized.

dble(x) returns an object of the same size and shape as x whose components are those of x, converted to double.

dcmplx(x [,y)]dcmplx(x) convert x to double complex type, dcmplx(x,y)=dble(x)+idble(y)

diag(x), diag(x,k) where x is a vector, returns a matrix whose main diagonal is x. The matrix will be n by n, where n is the length of x. diag(x,k), where x is a vector of length n and k is an integer, will return an `(n + |k|)` by `(n + |k|)` matrix with x on the kth diagonal (k may be negative; k = 0 is the main diagonal). The remainder of the entries are 0. If k is not an integer, it or its real part is truncated to integer. If k is not a scalar, its first component is extracted and used.

exp(x) returns the real, double or complex exponential of x, componentwise if necessary.

fft(x [,dim)] Fourier transform. x real or complex. If present, dim is the dimension over which the transform is taken for all values of the other subscripts. The transform length, n = length(x) or shape(x)(dim), can be any integer >0, but the method is most efficient when n is the product of small primes. For x complex, fft(x) returns z(j) = x .dot. exp(-2i*pi*j*iota (0,n-1)/n), j in range 0:n-1. For x real and n = 5 [6], fft(x) returns c0, c1, s1, c2, s2 [,c3], where cj = x .dot. cos(2*pi*j*iota(0,n-1)/n), and and sj = x .dot. -sin(2*pi*j*iota(0,n-1)/n). See also the inverse transform, ffti.

ffti(x [,dim)] Fourier inverse. For x real or complex, ffti(fft(x)) = x*length(x) for x one-dimensional, and ffti(fft(x,dim),dim) = x*shape(x)(dim) for any x with dimensionality >= dim. See also fft.

fit(x,y,n) fit(x,y,n) fits an n-th degree polynomial in x to y.

fromone(x) produces a value of the same type and shape as x, with its lowest subscript in each dimension set to 1.

float(x) returns an object of the same size and shape as x whose components are those of x, converted to real.

format(x,fw,nd,flg) returns a character string containing the formatted value of x. If fw is positive, the string length is fw; if fw is zero, the string has no leading blanks; if fw is negative, the string length is abs(fw) and the string is filled with leading zeros following the sign, if present. If nd is given, output is real with nd places after the decimal point. If flg is 1, output is fixed format; if 0, E- or D-format is used, depending on the type of x; if 2, E-format is used even if x is double precision

gather(x,index) gathers up a vector from source vector, x. x is a one dimensional array of type real, double, integer or complex. index is a one dimensional integer array which determines which elements are accessed. The output vector is the same type as the source vector. Its length is the same as the vector of indices. **EXAMPLE:**

```
real x(-2:2)=[-4,-2,0,2,4]
integer index(3)=[-2,0,2]
chameleon a=gather(x,index)
```

would result in:

```
a(1) = x(index(1)) = x(-2) = -4.00000e+00
a(2) = x(index(2)) = x(0)  =  0.
a(3) = x(index(3)) = x(2)  =  4.00000e+00
```

index(s,r) where `s` and `r` are strings, returns the position in `s` where `r` first appears as a substring, or zero if `r` is not found.

inf can have an arbitrary number of arguments of any sizes and shapes. inf returns a scalar which is the minimum value present amongst all the components of all the objects. Arguments must be of arithmetic type; only the real parts of complex objects participate.

int(x) returns x converted to integer, componentwise if necessary. If x is complex, int is applied to the real part. Conversion is by truncation.

iota(n), iota(m,n) where m and n are integer, returns a vector of length n-m, whose components, in order, are integers `m, m+1, m+2...,n`. If m is omitted it is assumed = 1.

land(x,y,...) can have an arbitrary number of logical arguments of any shapes. land returns a logical scalar which is true if every component of every argument is true.

length(a) returns the number of elements in a.

len_trim(s) returns the string length of string s without counting trailing blank characters.

load(a,n) returns a real vector with n components, the consecutive values in memory starting at address a. This function is useful for debugging.

log(x) returns the natural logarithm of object x, by components if necessary. See cos.

log10(x) returns the common logarithm of object x, by components if necessary. See cos.

lor(x,y,...) can have an arbitrary number of logical arguments of any shapes. lor returns a logical scalar which is true if some component of some argument is true.

max accepts two or more arguments and returns the maximum component by component. Scalars will be broadcast, but otherwise the arguments must have the same number of components. The result has the shape of the first non-scalar argument or is scalar if all the arguments are scalar. Only real parts of complex objects participate. See “sup”. 11

min does the same as max, but returns the minimum. See “inf”. 11

mnx(x,idim) returns the minimum index of array x. x can be of type real, double, or integer. The resulting type is the same as x. If idim is supplied, then the function is applied to each group of elements in x whose indices vary only in the i-th dimension. The output array produced is the same shape as x, except that the i-th dimension is removed. If idim is not supplied, then the function is applied to the entire array, resulting in a scalar output.

mod(x,y) returns the remainder after division of object x by object y. If x and y are not the same size and shape, y must be a scalar, which is then broadcast.

mxx(x,idim) returns the maximum index of array x. See “mnx(x,idim)”. 11

nint(x) returns the nearest integer to real object x. Similar to anint except for type of result.

ones(n) returns a vector of length n whose components are all 1, with a single scalar integer argument n. ones of more than one integer scalar argument returns an array of that shape whose components are the Kronecker delta.

outer(x,y) returns the outer product of objects x and y.

psum(x,idim) returns the partial sum of array x. x can be of type integer, real, double, or complex. The output is an array of the same type, size and shape as x. If idim is supplied, then the function is applied to each group of elements in x whose indices vary only in the i-th dimension. This value is stored in the output array element whose indices are the same as the indices of the input elements. If idim is not supplied, then the function is applied to the entire array.

ptp(x,idim) returns the peak to peak of array x (maximum value - minimum value). See “mnx(x,idim)”. 11

ranf(x) returns an object of the same size as x whose components are random numbers. These will be between 0 and 1 if x is integer, real, or double, on the unit circle if x is complex. See the chapter on Compiled Functions for additional documentation about ranf and its supporting routines *setranf*, *getranf*, *seedranf*, and *mixranf*.

rangex(x) where x is an array, returns a matrix whose rows contain the lower and upper subscripts for each dimension of x which is not of length 1. If x is scalar range, returns `[1,1]`.

EXAMPLES:

```
integer x(3,2:4,1,5), y(3,2:4,2,5)
rangex(x    # returns 4x2 matrix with rows
    # [1 3], [2,4], [1,1], [1,5]
rangex(y(,,,1:4))   # returns 4x2 matrix with rows
    # [1 3], [2,4], [1,2], [1,4]
```

rmsdv(x,idim) returns the root mean square deviation of array x. x can be of type real, double precision, or integer. The resulting type is the same as x, unless x is of type integer. In this case, the resulting type is real. If idim is supplied, then the function is applied to each group of elements in x whose indices vary only in the i-th dimension. The output array produced is the same shape as x, except that the i-th dimension is removed. If idim is not supplied, then the function is applied to the entire array.

rngbeg(rng, begindx) rng - the range (or array of ranges) whose beginning index (or indices) is to be returned.begindx - the integer value (values) to be used for any beginning index (or indices) whose value has been DEFAULTED.

This function returns the beginning index (or indices) of the given range(s) (argument rng) in which all the DEFAULTED fields (e.g. :10) have been replaced by the corresponding value

of argument begindx. If argument rng is an array, then argument begindx must be either a scalar integer or an array of the same length as argument rng.

EXAMPLES:

```
rngbeg(:8,2)    # returns 2
rngbeg(1:4, 3)  # returns 1
rngbeg([:8, 1:4, :7], 3)
    # returns vector [3,1,3]
rngbeg([:8, 1:4, :7], [1,2,3])
    # returns vector [1,1,3]
```

rngend(rng, endindx) rng - the range (or array of ranges) whose ending index (or indices) is to be returned.endindx - the integer value (values) to be used for any ending index (or indices) whose value has been DEFAULTED.

This function returns the ending index (or indices) of the given range(s) (argument rng) in which all the DEFAULTED fields (e.g. 2:) have been replaced by the corresponding value of argument endindx. If argument rng is an array, then argument endindx must be either a scalar integer or an array of the same length as argument rng.

EXAMPLES:

```
rngend(8:,15)   # returns 15
rngend(1:4, 3)  # returns 4
rngend([8:, 1:4, 7:], 3)
    # returns vector [3,4,3]
rngend([8:, 1:4, 7:], [11,12,13])
    # returns vector [11,4,13]
```

rnginc(rng), rnginc(rng, incindx) rng - the range (or array of ranges) whose stride (or strides) is to be returned.incindx - a value which is not used or checked. This argument need not be present.

This function returns the stride(s) of the given range(s) (argument rng). A second argument, incindx, is allowed but is not required (and is not used) in order to provide function RNGINC with an interface similar to RNGBEG and RNGEND. Defaulted values for range strides ar always 1. The first argument can be ny array in which case an array of increment fields is returned.

EXAMPLES:

```
rnginc(1:8,2)   # returns 1
rnginc(1:8) # returns 1
rnginc([1:8, 10:4:-1, 1:7:2])
    # returns vector [1,-1,2]
```

rngsetdf(rng, default_rng) rng - the range (or array of ranges) to be returned with any DEFAULTED fields replaced.default_rng - a range (or array of ranges) which has no DEFAULTED fields (increment fields are ignored). The corresponding fields will be returned in place of any DEFAULTED field in argument rng.

This function returns the given range(s) (argument rng) in which all the DEFAULTED fields have been replaced by the corresponding fields in the given default ranges (argument default_rng). If argument rng is an array, then argument default_rng must be either a scalar range or an array of the same length as argument rng.

EXAMPLES:

```
rngsetdf(:8, 2:10)  # returns 2:8
rngsetdf(2: , 1:22)
    # returns 2:22
rngsetdf(::3 , 1:15)
    # returns 1:15:3
rngsetdf([:8, 1:4,  7:], 2:15)
    # returns vector [2:8,1:4,7:15]
rngsetdf([:8, 1:4,  7:], [-1:5, 2:3, 3:9])
    # returns vector [-1:8,1:4,7:9]
```

rsum(x,idim) returns the partial sum of array x in reverse order. See “psum(x,idim)”. 11

shape(x), shape(x,n1,n2,...) shape(x), where x is an array, returns a vector that gives the shape of x (i.e., its ith component is the range of the ith subscript of x). All dimensions of length 1 are removed from x before the shape information is returned. If x is a scalar then shape(x) is 1. If the shape of x is one dimensional, shape(x) is a scalar giving the length.shape(x,n1,n2,...nk) returns x reshaped to the specified dimensions. It is an error if the length of x is not the product of those n1 through nk that are positive. If ni is negative, this is a so-called ”rubber index” signal. It indicates the number of repetitions of the data in the given dimension that are required. This is frequently used to create copies of data so as to match the shape of a higher-dimensional object with which it is used in arithmetic statements. The final number of elements in the result will be the product of the length of x with the absolute values of those ni that are negative.

EXAMPLES (reporting the shape):

```
integer x(3,2:4,1,5), y(3,2:4,2,5)
shape(x)    # returns [3,3,5]
shape(y(,,,1:4) # returns [3,3,2,4]
shape(0)    # returns 1
shape(iota(6))  # returns 6
```

EXAMPLES (changing the shape):

```
shape(x, 2, 3) # returns [[1,2], [3,4], [5,6]]
integer w = iota(3)
shape(w, -2, 3) # returns 2 by 3 matrix,
    # with each row = [1, 2, 3].
shape(w, 3, -2) \# returns 3 by 2 matrix,
    # rows = [1,1], [2,2], [3,3].
```

Suppose you have a variable x of shape (20, 35) and you wish to multiply each plane of y by it, where y is of shape (20, 35, 12). You would write: y * shape(x, 20, 35, -12)Note how you can read off the final shape of the result by taking absolute values.

sign(x,y) returns object x with the corresponding signs of the components of y attached to the components of x. If x and y are not the same size and shape, x must be a scalar, and it will be expanded into an object the same size and shape as y before the signs are attached, Both arguments must be integer or real, or double.

sin(x) returns the sine of object x, which must be in radians. See “cos(x)”. 11

sinh(x) returns the hyperbolic sine of object x. See “cos(x)”. 11

sngl(x) returns an object of the same size and shape as x whose components are those of x, converted to real.

sorti(&sortary, &sortidx, length) where sortary is an integer array to be sorted in ascending order and length is the number of integers to be sorted. The sorted list is returned in array sortary. Array sortidx is an output array of integers containing the permutation used to sort array sortary. The i-th value of sortidx is the index into the unsorted list of the i-th element in the returned sorted list.NOTE: an & is needed in front of both arguments sortary and sortidx since they return data.

spanl(start,stop,npoints) returns a list of floating point numbers logarithmically spaced, where start is the starting number, stop is the stopping number, and npoints is the number of points.

squeeze(x) array x is reshaped such that all dimensions of length 1 have been removed. If no such dimensions exist, then squeeze(x) is the same as x.

sqrt(x) returns the square root of object x. See “cos(x)”. 11

strchpat(s, oldpat, newpat) string substitute. Returns a string in which every occurrence of oldpat in s is substituted with newpat. if newpat is not specified, all occurrences of oldpat are removed.

strlen(s) returns the string length of string s.

struct accepts any number of arguments (including other structures) and returns a structure whose components are these objects. To access a component of a structure, one selects that component like an array component, by means of its subscript in parentheses.

substr(s,pos,len) returns a substring of s starting at the 1-origin index pos and is of length len.

sum(x,idim) returns the sum of array x. x can be of type integer, real, double, or complex. The resulting type is the same as x. If idim is supplied, then the function is applied to each group of elements in x whose indices vary only in the i-th dimension. The output array produced is the same shape as x, except that the i-th dimension is removed. If idim is not supplied, then the function is applied to the entire array, resulting in a scalar output.

sup can have an arbitrary number of arguments of any sizes and shapes. sup returns a scalar which is the maximum value present amongst all the components of all the objects. Arguments must be of arithmetic type; only the real parts of complex objects participate.

svd(x) svd(x)= singular value decomposition , structure (u, d, v) such that x = u *! diag(d) *! v', u, v unitary matrices, d vector of singular values.

tan(x) returns the tangent of the object x, which must be in radians. See "cos(x)". 11

tanh(x) returns the hyperbolic tangent of x. See "cos(x)". 11

tolower(s) converts a string from uppercase to lowercase.

toupper(s) converts a string from lowercase to uppercase.

transpose(x) transposes the matrix x.

trim(s) returns the string s with both leading and trailing blanks removed.

triml(s) returns the string s with leading blanks removed.

trimr(s) returns the string s with trailing blanks removed.

truerange(namex) where namex is string which is the name of an array or subscripted array. This function returns a matrix whose rows contain the lower and upper subscripts for each dimension of the array named by namex. If namex references a scalar range, returns `[1,1]`.The values returned by this function might be different than those returned by function rangex. Function truerange will return information about dimensions of length 1.

EXAMPLES:

```
integer x(3,2:4,1,5), y(3,2:4,2,5)
truerange("x")  # returns 4x2 matrix with rows
    #    [1 3], [2,4], [1,1], [1,5]
truerange("y(,,,1:4)")
    # returns 4x2 matrix with rows
    #    [1 3], [2,4], [1,2], [1,4]
```

trueshape(namex) where namex is a string which is the name of an array or subscripted array. This function returns a vector that gives the shape of the array named by namex (i.e., its ith component is the range of the ith subscript of x).The values returned by this function might be different than those returned by function shape. Function trueshape will return information about dimensions of length 1.

EXAMPLES:

```
integer x(3,2:4,1,5), y(3,2:4,2,5)
trueshape("x")  # returns [3,3,1,5]
trueshape("y(,,,1:4)")
```

type(x) returns the type of x as a string. Possible values are: "`integer`", "`real`", "`complex`", "`double`", "`logical`", "`character`", "`chameleon`", "`range`", "`function`", "`indirect`", "`structure`", "`word address`", and "`null`".

utype(t1,t2,...) Defines from one to ten user delimiters for use in COMMANDs. Each delimiter `ti` must be a string. Each call to utype creates a new table of user delimiters from scratch, and for subsequent reference in a COMMAND, they will be referred to by one of the digits 1, 2, ... , 9, 0 according to the order in which they occurred anmong the arguments of `utype`. `utype` is intended to be called as a subroutine, but if used as a function, it returns its last argument as its value.

vmax(x [,i)] with one argument, vmax(x) is the same as max(x), i. e., gives the maximum component of x, a scalar. vmax(x, i) performs the maximum over the ith dimension of x, delivering an array of one less dimension whose components are the maximum of x with those subscripts, as i varies. For example, if x(1,1) = 1, x(1,2) = 15, x(2,1) = 16, x(2,2) = 12, then vmax(x,1) is the vector [15,16] and vmax(x,2) is [16,15].

vmin(x [,i)] with one argument, vmin(x) is the same as min(x), i. e., gives the minimum component of x, a scalar. vmin(x, i) performs the minimum over the ith dimension of x, delivering an array of one less dimension whose components are the minimum of x with those subscripts, as i varies. For example, if x(1,1) = 1, x(1,2) = 15, x(2,1) = 11, x(2,2) = 12, then vmin(x,1) is the vector [1,11] and vmin(x,2) is [1,12].

where(cond,x,y) cond is an array or scalar of type logical. x and y each can be either a scalar or an array of the same length (but not necessarily the same shape) as cond; and their types can be either integer, real, double, or complex. Note: x and y do not have to be of the same type. If y is present, the output is the same size and shape as cond and its type is the more encompassing of x and y. For example, if x was of type integer and y was of type complex, then the output would be complex. All data will be coerced to the proper type before being stored into the output array. The value of the output array is calculated as follows. Those elements of the output array corresponding to true values of cond are set to the corresponding values of x (or set to x if x is a scalar). Those elements of the output array corresponding to a false value of cond are set to the corresponding values of y (or set to y if y is a scalar). If y is not present, then the output is a one dimensional array whose length is the number of true values in cond and whose type is the type of x. The values of the output array are those elements of x which correspond to a true value of cond. If x is a scalar, then all elements in the output array are set to x. Example: where([true,false,true],[1,2,3],0.0) evaluates to [1.0,0.0,3.0]

zcen(x) applies zone centering to the array specified.

CHAPTER

TWELVE

User-Defined Functions

12.1 Defining Functions

The user can define functions to perform some task not available in the built-in functions. At compile time, user-defined functions are translated into intermediate code, which is not executed upon completion of the function definition, but instead is stored. Later the function can be invoked, like a built-in, by executing code that calls the function.

The skeleton used for function definition is as follows:

```
FUNCTION name formalparams
     <stlist>
ENDF
```

`name` is any user identifier of 128 or fewer characters; it must not be a keyword. If a user variable or function with the same name already exists it is deleted.

`formalparams` is an optional parenthesized list of identifiers separated by commas. These identifiers are interpreted in `<stlist>` as associated with the actual parameter values passed at run-time. These names may be chosen arbitrarily; when the function is invoked variables with those names come into being at the highest level of the search stack. Thus, if a user-defined variable exists, and a function is called with that same name as a formal parameter, the user-defined variable will be inaccessible while in that function. If `formalparams` begins with a semicolon, or contains a semicolon in place of one of the commas, the arguments that follow the semicolon are optional. The user may call such a function with none, some, or all of its optional arguments. See “Functions With Variable Numbers of Arguments” on page 138 to see what happens in that case.

The `<stlist>` in a function is unrestricted, except that functions cannot be nested. An attempt to define a function inside a function or other structured statement is not allowed and will result in an error.

There is no provision for declaring the formal parameters and the function name. The formal parameters act like chameleon variables at call time, in that they assume all the attributes of the associated actual parameters. They can be coerced to a particular type by the built-in functions int, float, or cmplx, if the user wishes.

12.2 RETURN

An object (and hence a chameleon-like type, size, and shape) is associated with the function name by the statement

```
RETURN <lexp>
```

which returns control to the calling statement with the object <lexp> as the "value" returned by the function. A simple

```
RETURN
```

returns no value. If flow of control in a function drops to the ENDF, then a RETURN is automatically executed.

12.3 Local Variables

User variables can be declared inside a function. If they are, these variables are dynamic, and are allocated space when the function is executed. They are then deallocated upon RETURN. These variables can have the same names as other user variables, and if so, they supplant those variables as long as the function is at the end of the call chain. Because these variables exist only during execution of a function, they cannot be accessed from outside that function, and hence are strictly local to it.

Prefixing the declaration of a variable with the keyword GLOBAL creates a variable which is visible inside all functions and which replaces any currently existing variable with the same name.

12.4 CALL Is By Value

Actual parameters are passed by value, meaning that at run-time, their values are computed and passed to the function to be linked to the formal parameters. Thus, assignment of a value to a formal parameter in a function will not alter the actual parameter in the calling routine.

If a function is to have side effects (i.e., change the value of some existing variable), then this may be done by accessing the variable globally: a variable is accessible to a function if it has been declared outside all functions (or is predefined), and if no formal parameters or local variables within the current function have that same name. Another method is to make the formal argument the name of the array, and use an INDIRECT variable to reference it (See "Indirect Variables" on page 17.)

A function can be invoked in either one of two ways. The first is exactly the same as for built-in functions: as an operand in an expression (function name followed by expressions for its actual

parameters in parentheses). The number of actual parameters must agree with the number of formal parameters (possibly zero) declared in the FUNCTION header.

The second way to invoke a function is to use the CALL statement

```
CALL name actualparams
```

where `name` is the function being invoked. `actualparams` is optional; it can be absent if name was declared with no parameters, and present (with the same number of parameters) when name was declared with parameters. Whether or not name returns a value is irrelevant to the CALL statement, since the value is discarded. Presumably, CALL only makes sense if used with a function that has side effects, say giving values to global variables, displaying values, or running physics packages.

If a function has no formal parameters then it can be called in any of the following ways:

```
CALL name
CALL name()
name()
name
```

but of these only `name()` can be used in an expression, and only `name` or `name()` return a value.

The meaning of a name is decided at execution time, so it is acceptable to define a function that calls a second function that has not yet been defined, as long as the second function is defined before the first function is executed. If this is not done, an error message will be received to the effect that the name does not exist. A function MAY call itself; logic to prevent infinite recursion is your responsibility.

12.5 Examples of User Functions

As an example of function definition, the following function computes the square root of its argument (if real and positive) within a specified tolerance:

```
FUNCTION myroot (x,eps)
REAL y= 1.,eps1 = max(abs(eps),1.e-12)
IF (x<0) THEN
     REMARK "myroot called with negative value";x
     RETURN 0.0
ENDIF
IF(x = 0) THEN RETURN 0.0
DO
     y = .5*(y + x/y)
UNTIL (abs(y*y-x) <eps1)
RETURN (y)
ENDF
```

The tolerance of the result is measured with eps. Note that 0.0 is returned if the actual argument was negative. In addition, an error comment is printed and x is displayed. If x is not negative, then either 0.0 is returned or else iteration proceeds until the desired tolerance is met.

The following function computes the value of n factorial. Note that this function calls itself recursively, which is allowed in Basis.

```
FUNCTION nfact (n)
IF (n<0) THEN
     REMARK "negative factorial not defined"
ELSEIF (n = 0)
     RETURN 1
ELSE
     RETURN n*nfact(n-1)
ENDIF
ENDF
```

CHAPTER

THIRTEEN

Compiled Functions

Certain packages, including the Basis parser, contain compiled functions and subroutines. These are modules written in Fortran or assembler which have been compiled and loaded into the executable program. Basis has the ability to execute some of these functions and subroutines in a way similar to the way Basis executes built-in and user-defined functions.

The difference between a function and a subroutine is that a function returns a value, while a subroutine does not. In what follows we will simply use the name “function” to mean “function or subroutine”.

In order for the function to be executable from Basis, the function must be listed amongst the variables of the package. That is, the author of the package had to incorporate a description of the function into his or her variable description file, which is part of the process of making a Basis code. The function will be listed with a “template” for its calling sequence, which will consist of a parenthesized list of arguments with optional types attached with a colon to the name, such as:

```
blah(x,y,z:integer,w:string) complex
```

This means that blah is a function that takes 4 arguments, the first two of type real (by default, since they have names that begin with letters other than i through n), the third one of type integer, and the fourth one of type “string” which means a character string of any length up to 500. The function blah returns a complex value.

The functions that are declared in the parser package are described in this manual. (33) For other packages, consult the documentation supplied by the author or poke around with the list command until you find some.

Compiled functions can be of type integer, real, double, complex, logical, or character*(n). Arguments to compiled functions can be of the same types. Currently, arguments to compiled functions cannot be the names of functions of any type. Compiled functions only return scalar values.

A compiled function that modifies one of its input arguments may be dangerous. Basis has no way of checking the length of an array expected by a compiled function, and a call to a compiled function that modifies a location not supplied in the call will typically cause Basis to crash. Also note that functions are called by value, and hence the modified argument is not accessible after the function returns (but see below). However, used properly, compiled functions can be a very powerful tool.

13.1 CALLing By Address

A (possibly subscripted) variable can be passed to a compiled function by address. (See “Indirect Variables”, 4.7 for the equivalent method for user-defined functions.) In this case modifications which the function makes to the array WILL change the original. To do this, precede the name with an ampersand in the calling sequence. For example, suppose zero(x,n) is a routine which sets the first n components of x to zero. Then

```
real x(10) = iota(10) ; call zero(&x,5); call zero(&x(7),1)
```

will result in x containing `[0.,0.,0.,0.,0.,6.,0.,8.,9.,10.]`.

Only compiled functions can be passed an argument by address. If the variable being passed by address is of any character type, it cannot be subscripted. When a variable is passed by address no type conversion is done.

CHAPTER

FOURTEEN

Defining Your Own Commands

14.1 The COMMAND Statement

```
fname COMMAND arglist
fname command_spec arglist
```

A formal description of this syntax will be developed step-by-step in this section. Informally, the COMMAND statement will cause function `fname` to be called with arguments `arglist`. This capability can be used in conjunction with the macro facility to define your own blank and/or comma delimited "commands". It is also possible to specify other delimiters between arguments, or even to define your own. In the following example

```
mdef mycommand = myfunction command mend
mycommand arg1, arg2
```

a command called `mycommand` is defined and is used. The above usage of this command (`mycommand arg1, arg2`) will cause function `myfunction` to be called with arguments `arg1` and `arg2`.

The formal definition of the syntax for argument specification is as follows. The first syntax form is a function name `fname` followed by the word `COMMAND` (or `command`) followed by a comma and/or blank delimited argument list – WITHOUT parentheses. This form of the COMMAND statement expects `arglist` to contain only expressions, i.e. all arguments are first evaluated. Thus all strings must be quoted.

The second form is identical to the first form except that instead of entering the word "`command`", you now enter a `command_spec` which is the word "`command_`" (note the underscore) immediately followed by a type specification. This specification is a series of `s`'s, `S`'s, `e`'s, and/or `E`'s, optionally followed by a parenthesized series of `s`'s, `S`'s, `e`'s, and/or `E`'s at the end. (This second form also allows you to specify what delimiters you wish as defaults between arguments, and any special delimiters between specified arguments. We will postpone the question of delimiters until a later section.) This specification allows you to have arguments which are either unquoted strings (`s, S`) or expressions (`e, E`), or a combination of both. The first letter of the specification defines the type of the first item in the COMMAND `arglist`, the second letter, the second item,

etc. The s denotes an unquoted string and an e an expression. The upper case letters S and E also denote unquoted strings and expressions, except that macro expansion will be suppressed; thus the macro-suppressing brackets "{}" are not necessary in the corresponding arguments. The letters of the specification within the ()'s (if any) are used repeatedly until the end of the command list (i.e. (se) is sesese...). If no ()'s are present in the specification, then the last letter is used repeatedly until the end of the COMMAND arglist (i.e. "se" is the same as "seee...").

The following example illustrates defining a new "command" called gotoit which expects a series of expressions as arguments. A second "command" called dothis is also defined, which expects a series of name and expression pairs.

```
define gotoit   f COMMAND
define dothis   g command_(se)
gotoit a,4+5,c
dothis x 6+1 y 9+3
```

The above example is equivalent to

```
call f(a,9,c)
call g("x", 7, "y", 12)
```

This is accomplished in two stages. First, "gotoit a,4+5,c" is expanded to "f COMMAND a,4+5,c". The COMMAND statement then evaluates expressions a, 4+5, and c and calls function f with the resulting values. In the next example, "dothis x 6+1 y 9+3" is expanded to "g command_(se) x 6+1 y 9+3". The "command_(se)" statement treats x and y as strings and evaluates expressions 6+1 and 9+3. Then function g is called with the resulting values. It should be noted that f and g can be any type of function such as a Basis function, built-in function, or compiled function or subroutine.

The above examples show that macros and COMMANDS can interact to provide a powerful tool. Besides being able to use a COMMAND in a macro, you can also use macros in a COMMAND arglist. If the arguments are specified by lower case e or s, then macros used in an argument are expanded, unless they are protected by curly brackets { }'s (or are enclosed in quotation marks). Macros are not expanded in arguments specified by upper case S and E, so it is not necessary to use the somewhat clumsy bracket notation to suppress macro evaluation.

Some examples follow which show two things – different ways to delimit a COMMAND arglist, and the usage of macros in an arglist.

```
define expr (6+2)
define x   str1
mdef y = str2 mend
mdef z = str3 mend
echo COMMAND a, (6+2)/2, c      # comma delimited
echo COMMAND a  expr/2  c       # blank delimited
echo COMMAND a expr/2,          # mixed delimited
```

```
              c                 #   continuation of arglist
echo COMMAND a {expr}/2 c       # one way to suppress macro
echo COMMAND_eEe a expr/2 c     # another way to do it
echo command_s str1, str2, str3 # comma delimited
echo command_s x y z            # blank delimited
echo command_s x y,             # mixed delimited
              z                 #   continuation of arglist
```

The first set of COMMAND statements is equivalent to `call echo(a,4,c)`, the second to `call echo(a,expr/2,c)` (one hopes that there is a variable named `expr` which is defined and has a value), and the third set to `call echo ("str1", "str2", "str3")`. Note that in the third and eighth COMMAND statements that both commas and blanks were used as delimiters in a single statement. A comma at the end of any COMMAND line signifies that the COMMAND `arglist` continues on the next line.

WARNING: It should be noted that

```
define expr 6+2
define exprnew  7 +1
mdef mystring = this is a string mend
echo COMMAND a, expr/2, c
echo COMMAND a, exprnew, c
echo COMMAND a, 7  +1,  c
echo command_s this is a string
echo command_s mystring
```

is equivalent to

```
call echo(a, 7, c)
call echo(a, 7, +1, c)
call echo(a, 7, +1, c)
call echo("this", "is", "a", "string")
call echo("this", "is", "a", "string")
```

The expression "`expr/2`" is equal to seven since this expression expands to "`6+2/2`". Macro writers should remember to enclose any expressions in parentheses, such as "`(expr)/2`", if they wish their expression to be evaluated before any other operations are performed.

The rest of the preceding examples deal with delimiting issues. Blanks and commas are the default delimiters between `command` arguments. Thus the blanks between the "`7`" and the "`+1`" and between the words in "`this is a string`" are considered delimiters, even if these blanks appear in the middle of a macro definition. If you have an expression with blanks that you wish to be considered one expression, then enclose the expression in parentheses ()'s. If you have an unquoted string with blanks (or commas) that you wish to be considered one string then enclose the entire string (or just the blanks or commas) in quotation marks (") or protection brackets { }'s. (The other method of specifying delimiters other than the defaults is discussed in a future subsection). Thus

```
define expr (6+2)
define exprnew (7 +1)
mdef mystring = {this is a string} mend
echo COMMAND a, expr/2, c
echo COMMAND a, exprnew, c
echo COMMAND a  (7  +  1) c
echo command_s "this is a string"
echo command_s mystring{ and here is some more}
```

is equivalent to

```
call echo(a, 4, c)
call echo(a, 8, c)
call echo(a, 8, c)
call echo("this is a string")
call echo("this is a string and here is some more")
```

14.2 Changing the Default Type of a COMMAND Argument

Suppose you define a new "`command`" `helpme` which expects a series of unquoted strings. Now what if someone using your "`command`" `helpme` wants to use an expression to calculate the value of a string? This user can override your default type specification of "`command`" `helpme` by typing a caret "^" (the user can have blanks following the caret) in front of the arguments whose type she wants to change. The caret will change strings to expressions and expressions to strings (it will not change the case, i. e., `s` becomes `e` and `S` becomes `E`).

EXAMPLES:

```
define helpme  g command_s
define exprs   f command
character*5 root = "mynam"; integer i=5
helpme x y z
helpme x ^ root//format(i,0) z
helpme x ^ (root // format(i,0)) z
exprs 1+2 3+4
exprs ^ abc ^ 3+4
```

The above examples are equivalent to

```
call g("x", "y", "z")
call g("x", "mynam5", "z")
call g("x", "mynam5", "z")
call f(3, 7)
call f("abc", "3+4")
```

Blanks are considered delimiters. The only difference between the second and third `helpme` commands in the above example is the spaces surrounding the // operator and the parentheses surrounding the expressions. Because of these blanks the parentheses are needed.

14.3 Specifying Other Delimiters in a COMMAND Statement

There is an additional COMMAND syntax which allows one to specify default delimiters (other than blanks and commas) for the entire command and to vary these delimiters between individual pairs of arguments; it also allows users to define their own delimiters and to specify those. Users can define their own delimiters by means of the new builtin function `utype`. One calls this function with a list of from one to ten strings, which are then entered into a table in order. For example

```
call utype("=","with","?","by")
```

will define user delimiter number 1 as `"="`, number 2 as `"with"`, and number 3 as `"?"`. There are no other user delimiters. (Each subsequent call to `utype` redefines the user delimiter list to whatever its arguments are.) When specifying user delimiters, the number of the delimiter (1 through 9, with 0 representing the tenth) is used.

Delimiters available from the system are blank and comma (which are always the defaults if nothing else is specified), the `at` symbol , and the equal sign `=`. These are denoted (respectively) by `W` (for "whitespace") `C`, `A`, and `Q`. The lower case letters `w`, `c`, `a`, and `q` are used to denote the suppression of a particular delimiter.

The syntax is that `command_` is followed by a string including one or more of the delimiter characters and 's'/'S', 'e'/'E' characters, with an optional set of parentheses, as follows:

```
command_<string1><string2>(<string3>)
```

where at least one of `<string1>`, `<string2>`, and `(<string3>)` must be present, and `<string2>` and `(string3>)` (if present) must begin with one of the argument designators `e, E, s,` or `S`, and:

<**string1**> (if present) is a string of delimiter characters and specifies the default delimiters between the arguments of this command. This may consist of up to four of the letters W/w (white space is/is not a default delimiter), C/c (comma is/is not a default delimiter), A/a ("at" is/is not a default delimiter), or Q/q ("equals" is/is not a default delimiter); and any combination of digits standing for user delimiters. The order is unimportant. If <string1> is absent it defaults to 'WC'.

<**string2**> (if present) consists of a list of the letters s (S) and/or e (E), each optionally followed by delimiter characters as enumerated above. Delimiter characters in between argument characters are used to modify the default delimiters between those two arguments only. They may be used either to specify additional delimiters or to enable (or suppress) one of the four standard ones.

(<string3>) (if present) consists of a parenthesized list of argument and delimiter designators as in <string2>. The parentheses mean to repeat <string3> as often as necessary to include the rest of the arguments. If (<string3>) is not present, then the last argument designator in <string2> will be applied repeatedly, if necessary, to cover all specified arguments. If <string2> is absent as well, then all arguments will default to expressions with macro expansion enabled (e). Note that (<string3>), if present, is required to contain at least one argument designator, and it must be the first character after the "(".

Here are some examples. We shall assume in what follows that utype has been called to set up user delimiters "?", "with", "%", "by", in that order.

```
f command_wse the first argument, ( 6 + 8) * 43 ,88
```

is equivalent to

```
call f(" the first argument",(6+8)*43,88)
```

Note that the initial w suppresses the use of white space as delimiter, so the only default delimiter left is the comma. White space is gathered as part of the string argument, but has no significance in the expression arguments.

```
define name sam
define macpak specialvar
g command_s2SQe1(e1) name with macpak = 4 ? 3 ? 2 ? 1.56
```

is equivalent to

```
call g("sam","macpak",4,3,2,1.56)
```

Here the first string, name, is expanded as a macro, while the second, macpak, is not, because it was specified with a capital S. The number 2 user delimiter with was used between the first two arguments, then = as specified by Q, and then the rest of the delimiters are number 1 (?) as specified by the repeated designation e1 in parentheses. Although white space is a default delimiter throughout this command, note that white space which surrounds non-blank delimiters is ignored. Indeed, white space is necessary surrounding with because it would not be recognized as a separate token otherwise.

```
h COMMAND_SQe2e4e name = 6 with 18 by 2
```

is equivalent to

```
call h("name",6,18,2)
```

This time name is not expanded because it was specified by upper case S. Equal (Q) and user delimiters with (2) and by (4) were specified as additional delimiters between the second and third, and third and fourth, arguments, respectively. The occurrence of 2 and 4 only ALLOWS the with and by to be used as delimiters. Comma and white space are still valid delimiters unless specifically suppressed by lower case w and c.

14.4 No Delimiters at All: the COMMAND_L

The COMMAND_L is a special construct not previously mentioned. It allows the entire line following the 'command_l' to be gathered as one argument; nothing is accepted in any way as special except the end of the line. Thus in effect 'command_l' grafts quotes onto the beginning and end of the rest of the line. For example,

```
parsestr command_l integer x = shape(iota(27),9,3); x #define x
```

is the same as

```
call parsestr(" integer x = shape(iota(27),9,3); x #define x")
```

whereas

```
parsestr command_wcs integer x = shape(iota(27),9,3); x #define x
```

is equivalent to

```
call parsestr(" integer x = shape(iota(27),9,3)"); x #define x
```

The action is quite different; in the first case, the entire rest of the line is picked up and enclosed in quotes; in the second case, argument gathering stops with the semicolon. (This is intended to illustrate that either a semicolon or a pound sign will normally cause the gathering of COMMAND arguments to cease. COMMAND_L is intended to be a way of getting around this.) In the second case, `x` may be an unknown variable, globally, or if known, will almost certainly be different from the array defined locally by `parsestr` and then lost upon return. To illustrate this, here is a sample Basis run:

```
Basis> character*4 x = "abcd"
Basis\> parsestr command_l integer x = shape(iota(27),9,3); x
x                   shape: (9,3)
 row col =       1  2  3
 1:     -        1 10 19
 2:     -        2 11 20
 3:     -        3 12 21
 4:     -        4 13 22
 5:     -        5 14 23
 6:     -        6 15 24
 7:     -        7 16 25
 8:     -        8 17 26
 9:     -        9 18 27
Basis> x
x        = "abcd"
Basis\> parsestr command_wcs integer x = shape(iota(27),9,3); x
x        = "abcd"
```

Notice how neither command destroys the global value of `x`; but the first command prints out the local value (thus showing that its entire argument has been enclosed in quotes), while the second prints out the global one (showing that the semicolon has been recognized as a statement separator, and hence as the end of the command argument).

With some care, the `l` argument specifier may be used in combination with other arguments, observing the following constraints: `l` (or `L`, which has exactly the same meaning) must be the last letter present in <`string2`>, and (<`string3`>) can not be present. This ought to be obvious after a moment's thought, because if `l/L` suppresses the recognition of all delimiters, then there is no way to collect an argument following the one it specifies.

For example,

```
f command_eel 6+8*4 5-3*7+2 string ; x # argument
```

is equivalent to

```
call f(6+8*4,5-3*7+2,"string ; x # argument")
```

CHAPTER

FIFTEEN

The Search Stack

Basis designates one package as the “current” package; this allows the user to specify one package to be searched first during display of values or listing of variables. The parser itself is a package named “par”. Basis begins with “par” as the current package, but the user can designate a new current package by entering PACKAGE pkgname. If desired, the user can specify several packages in order to create a “search stack”. The commands POP and PACKAGE pkgname can be used to manipulate the search stack. The current package is the package at the top of the stack. “par” is always present at the bottom.

The search stack is initialized by the program author. The command `list packages` can be used to see the current search stack. Consult the documentation for your particular program.

In searching for a variable or function name, Basis searches in the following order: first, the local variables and formal parameters of the currently executing function; second, the user-defined variables and functions; next, the packages on the search stack are examined in order. This stack always ends with the variables of the parser itself, the package called “par”. The searching is done at execution time, not compile time.

CHAPTER

SIXTEEN

Package Control Statements

```
package pkgname
```

places `pkgname` at the top of the search stack, making it the current package. If `pkgname` was already in the search stack, it is moved to the top.

The routine `parpop` removes the top element of the search stack, making the next element the current package. If the stack has been reduced to `par` there is no effect.

CHAPTER

SEVENTEEN

The CTL Package

Some Basis programs use a special package named `ctl` to control the execution of physics packages. You can check if `ctl` is present in a program with the command `list packages`. If it is, commands named `run`, `generate`, `step`, and `finish` will be defined for you. These commands are in chapter **??** in *The Basis Package Library* document.

CHAPTER

EIGHTEEN

Removing Functions and Variables

Users may sometimes want to delete one or more user-defined functions or variables that are no longer needed. The FORGET statement allows this to be done. A simple

```
FORGET
```

wipes out all user-defined functions and variables, and releases the space occupied by them so that it can be reused.

```
FORGET name1, name2, ...
```

where name1, name2, are the names of a user-defined functions or variables, deletes those names and releases the corresponding space.

If the user want to delete a macro, he should use the UNDEFINE command.

If the user wants to protect user-defined functions and variables made up to this point from future FORGETs he can enter:

```
call protect
```

CHAPTER

NINETEEN

LIST Command

One important feature of Basis is that it knows about the variables in the different packages and can tell the user about them. The author of a package organizes the variables into “groups”. The command LIST is used to display information about variables, groups, and packages.

```
LIST [ name ]
```

where name is the name of a variable, group, or one of the keywords packages, macros, groups, variables, or functions.

LIST LIST (with no argument) displays a help package for the LIST command.

LIST macros LIST macros displays a list of the macros that have been defined.

LIST packages LIST packages displays a list of the packages in Basis, giving the name of the package, a short description, and its current status.

LIST pkg.variables displays a list of the variables (and functions) in that have been declared in package `pkg`, sorted by group. If `pkg` is omitted it defaults to the user-defined variables.

LIST pkg.groups LIST groups displays a list of the groups in the package `pkg` with a short description. A group is a group of variables that the author of a package has designated as logically related to one another. If `pkg` is omitted it defaults to the user-defined variables.

LIST pkg.functions LIST FUNCTIONS displays a list of functions in the package `pkg`. If `pkg` is omitted it defaults to the user-defined functions.

LIST Group LIST Group displays information about all the variables in the group named Group. Group must be entered with correct case; at least the first character will be upper case. The name of the group may be abbreviated to any unique prefix. One special group is the group User, which contains all the variables and functions declared by the user. When a user function is executing, there is a special group named `Locals_fname` where fname is the name of the function. This group can be edited or listed while the function is executing or while any function called by it is executing. If fname calls itself, only the most recent incarnation can be viewed in this way.The parser package par contains two groups `Builtin_Functions`

and `Compiled_Functions`. LIST Builtin displays a list of the built-in functions such as sqrt. (See Chapter 11, “Built-in Functions” for a full description of the available functions.) In general, all the Fortran intrinsics are available, in generic form. Thus for example, one can use sqrt(x) to get the appropriate square root of x whether x is integer, real, double, or complex. LIST Compiled displays a list of compiled parser functions.

LIST name LIST name displays information about the name, including type, length, location, whether or not it is dynamic, its dimension, and a comment about it made by the package writer, and its attributes. If name is the name of a function, some different information about it is displayed. If name is the name of a macro then the definition of the macro is displayed along with whether it was declared with or without arguments.The user may prefix the name by a package name and a period, e.g., vf.sigcoef where vf is the package name and sigcoef is a variable name.Variables local to a function are visible ONLY when the flow of execution is currently in that function; such variables are NOT visible in functions that are called by it. There is a way to see such variables, however, for debugging purposes: include the statement `Locals_fname` in the function.

CHAPTER

TWENTY

Obtaining and Setting Scalar Values

The following functions accept an argument of type character, which should contain the name of a variable; if the variable is a scalar of the right type, the function returns its value. If it is the wrong type, or is not a scalar, or does not exist, then `kaboom` is called. Otherwise, these functions can be used in any expression.

```
ibasis(s: string) integer function
--return an integer value

rbasis(s: string) real function
--return a real value

dbasis(s: string) double function
--return a double precision value

cbasis(s: string) complex function
--return a complex value

lbasis(s: string) logical function
--return a logical value

sbasis(s: string) character*MAXSTRING function
--return a string value.
```

Since the value returned by `sbasis` is `MAXSTRING` characters long, there may be lots of extraneous blanks on the end. To get rid of these, you could define a macro, which would trim trailing blanks before returning the value. Here's an example without the macro, using `sbasis` just as written:

```
Basis> character *20 s = "Short String"
Basis> sbasis("s")
sbasis   = "Short String"
Basis>
```

Note the long string of unnecessary and distracting blanks. On the other hand, with the following macro definition,

```
mdef sbasis() =  trim({sbasis}($1))  mend
```

the following exchange would result:

```
Basis> character *20 s = "Short String"
Basis> sbasis("s")
trim     = "Short String"
Basis>
```

These next subroutines are complementary to `ibasis`, `rbasis`, etc., described above. They accept a string and a value of the appropriate type; if the string contains the name of a scalar variable and the type is right, then the subroutine assigns the value to the variable. The routines are:

```
sibasis(s: string, v: integer) subroutine
--set an integer value

srbasis(s:string, v: real) subroutine
--set a real value

sdbasis(s: string, v: double) subroutine
--set a double precision value

scbasis(s: string, v: complex) subroutine
--set a complex value

slbasis(s: string, v: logical) subroutine
--set a logical value

ssbasis(s: string, v: string) subroutine
--set a string value.
```

In the case of `ssbasis` the assignment follows the usual FORTRAN rules: if the source string is too long for the destination, it will be truncated from the right. If it is shorter than the destination, then the destination will be blank-filled on the right.

CHAPTER

TWENTYONE

Help and News

The function `news` displays information about recent changes. The files news and newslog inside `basis` contain the recent and cumulative news respectively. To invoke news, just enter `news` at the prompt.

The function `help` displays information about how to get further help on Basis and/or the physics package you are using.

Both `help` and `news` are simply compiled functions being executed by the Basis interpreter; these no longer are keywords.

CHAPTER

TWENTYTWO

Input, Output, and External File Access

22.1 Reading Basis Code From a Text File

```
READ filename
```

shifts input from the current source to filename, a text file created in advance by the user that contains Basis statements. After all statements in filename have been read, parsed, and executed, input resumes from the current source, including the rest of the line on which the READ command occurred. READ commands can also occur in files, to a depth of up to ten files. As the commands are read, they are displayed on the terminal unless an

```
echo = no
```

or

```
echo = logonly
```

command has been given. It may be desirable to put comments in the input files; this can be done by prefixing them with a pound sign (#). Everything on a line after a pound sign is taken as a comment. If `echo = no`, the user can still use the REMARK "message" command to display progress reports on the terminal.

If filename is not in the current working directory, then Basis looks for filename in the following default directories in the order specified:

1. The directory from which the Basis source is being executed.
2. The directory specified by the environment variable WRK (if defined).
3. The directory specified by the environment variable HOME (if defined).
4. The directory $BASIS_ROOT/include (if environment variable BASIS_ROOT is defined.)

The Basis functions pathadd and pathrm may be used at runtime to add or remove other directories to and from the search path. pathadd (”directory-name”) adds ”directory-name” to the top of the list and pathrm (”directory-name”) removes ”directory-name” from the list.

In addition, the Configure file has keywords codefile and path, which allow the user to specify at compile time additional search paths. The reader is referred to *Writing Basis Programs: A Manual For Authors*, page **??**, for a discussion of these keywords.

Basis displays an error message if it fails to find the specified file.

IMPORTANT: The READ command should not normally be used in combination with Basis compound statements such as DO loops, IF-THEN statements, FUNCTION’s, and so on. A READ in the middle of a compound statement is not executed until the compound statement has finished.[1] If more than one READ occurs inside a structured statement, they will be executed after that statement completes, but in the reverse order of their occurrence. For example, the following code sequence:

```
if (x == 3) then
   read a3
   read a4
   << "I just finished reading a3 and a4."
endif
```

is executed in the order as if it were written:

```
if (x == 3) then
   << "I just finished reading a3 and a4."
   read a4
   read a3
endif
```

Thus, while it is not illegal to put READ statements inside compound Basis statements, it is almost never correct.

If you wish to skip the first n records of the file, enter:

```
nskipr = n
```

before your READ command. Basis will automatically reset `nskipr` to 0.

[1] More precisely, *execution* of a READ statement simply opens the file and pushes its descriptor onto the stack of files from which Basis will read next. Processing new text from the next file does not actually begin until the (largest) enclosing, compound, statement has finished.

22.2 Resuming Reading

```
RESUME [n] [filename]
```

RESUME resumes reading a file after a crash has occurred while reading some file. If filename is entered Basis resumes reading from filename, otherwise it resumes in the file where the crash occurred. If n is entered reading starts at line n, rather than with the line where the crash occurred.

22.3 Printing Messages on the Terminal

```
REMARK "message"
```

displays message on the terminal. This can document progress in executing the file when the variable echo equals no. If v is a character variable or expression, REMARK v prints the contents of v to the terminal.

22.4 Changing the Destination of Basis Output

```
OUTPUT TO filename
```

causes most subsequent output to go to the file filename. OUTPUT TO TTY closes the output file and returns Basis to writing output on the terminal. OUTPUT TO GRAPHICS sends the output to the plot file.

CHAPTER

TWENTYTHREE

The Stream I/O Facility

23.1 Introduction to Stream I/O

Stream input and output features are available in Basis. The user may read input from an existing file. The user may create an output file, or send output to a plot or terminal. This section discusses how each of these tasks may be accomplished. First we show how to use the function `basopen` to open an input file or create an output file. Then we introduce the Basis input operator >> and show how it can be used to read data from an input file. Next we introduce the Basis output operator <<, illustrating how to send output to an output file, to a terminal or to a plot. We then discuss how the user can format output by using the function format. Lastly, we show how to use the subroutine `basclose` to close files that have been opened using `basopen`.

23.2 Opening and Creating Files

To read input from an existing file or create an output file, the user must first open the file by calling the function `basopen`. Any attempt to read input from a file or send output to a file without first using `basopen` to open that file will result in an a semantic error. It is not necessary to call `basopen` when sending output to a terminal or plot.

The function `basopen` is an integer function that accepts two arguments: a filename and a specification. It returns a unit specifier which is used to direct output to and retrieve input from a specific file.

The general form of the function call is:

```
unit = basopen(filename, filespec)
```

where:

unit is a unique unit specifier whose value is set by `basopen`. The unit specifier is used in the input and output commands to specify the file being used. Once the unit specifier has been assigned a value by `basopen`, it should not be altered.

filename is the name of the file being opened. Its length can be up to 128 characters.

filespec filespec should be ”r” or ”w” or ”i”.A sequential formatted file is created, opened or inquired about. `kaboom` is called if anything goes wrong.When rw = ”r”: `basopen` searches for the file in the current directory and then in any lib libraries specified in the variable path (or in directories on Unix systems).If a file must be found in a lib library on NLTSS a copy is made to a temporary file with a different name. This file is destroyed when the program terminates.No copies are made when opening files in UNICOS or SUNOS directories.When rw = ”w”: if an error occurs, this file will be closed.When rw = ”i” the file is not opened; instead, OK or ERR is returned, indicating whether or not filename could be opened for reading.

It is possible to have several files open at once, provided each file has its own unit specifier. This means the user should use a different integer each time he or she opens a new file. Here are a few examples of how to create output files and open existing input files:

Creating output files:

```
outunit = basopen("newfile", "w")
number  = basopen("one", "write")
junk    = basopen("asdf","WRITE")
mine    = basopen("myfile",  "W")
```

Opening input files:

```
infile = basopen("mydata",  "r")
myin   = basopen("data1","read")
x      = basopen("input",   "R")
in     = basopen("test", "READ")
```

Note that abbreviations may be used for `"read"` and `"write"`. The two most important things to remember are that the unit specifier is an integer and must not be modified by the user once it has been set by Basis in `basopen`.

`basclose(unit)` should be used to close files opened with `basopen`.

23.3 The Input Operator >>

Basis stream input must be read from an existing file or from the terminal. If input is from a file, then the file must have already been opened using the function `basopen` (as described above) before any input can be read. Once this has been done, the input command may be used to retrieve input from the open file.

As a general rule, Basis stream input can read files created by Basis stream output (see See “The Output Operator <<” on page **??**.) There is one important exception to this rule. Double precision

numbers with three-digit exponents are sent out without a “D” preceding the exponent, regardless of formatting (FORTRAN does the same thing). The stream input lexical analyzer will therefore see the double precision number as a real (the mantissa) followed by an integer (the exponent). Although this may seem at first glance undesirable, this is precisely the behavior of FORTRAN on unformatted input.

Stream input from files may be done in two modes, noisy and non-noisy. Non-noisy mode might be used to read real, integer, and double precision numbers (complex numbers are not presently supported) from a text file which was produced as output by a FORTRAN program. This file may contain the numbers in tabular form and might include explanatory text and other non-numeric information. In non-noisy mode, all non-numeric items in the file are considered to be “noise” and are ignored. In noisy mode, the so-called “noise” is not ignored, but will be read in as character strings. WARNING: character string stream I/O will only work if the strings contain no imbedded blank characters except for trailing blanks. Even trailing blanks can’t appear if character arrays are being processed. Noisy mode will be explained in more detail below.

The Basis input statement consists of a unit specifier and one or more input variables or arrays. The general form of this command is:

```
unit >> var1 >> array1 >> var2 >> var3 ...
```

where

`unit` indicates the file from which the input is coming. It is the unit specifier which was returned by `basopen` when the file was opened. If the unit specifier is omitted, the terminal is assumed, and an input prompt will appear there.

`var1, etc.` indicates the variables and arrays to which the input values are being assigned. The user may input array elements or entire arrays.

Exactly how the input is assigned to the variables depends on the setting of the built-in variable “`noisy`” (whose default value is “`no`.”) This value may be set to “`yes`” or “`no`” by assignment, thus:

```
noisy = yes
```

When “noisy” is “no,” then numerical tokens (i. e., legal FORTRAN integers , reals, and doubles) are extracted from the input in the order that they occur and are assigned to the variables (“var1”, “var2”, etc.) also in the order of occurrence. All other characters in the input, which are either delimiters (currently, spaces and commas) or are not FORTRAN integers, reals, or doubles are treated as “noise” and ignored. The input command extracts the next available numbers from the input file, even if it must go to subsequent lines of the input file to do so.

In this mode all input variables and arrays must be numeric (integer, real, double). If an attempt is made to read to a non-numeric variable or array, then an error diagnostic occurs and the input file is closed (assuming it is not the terminal). If an attempt is made to read past an end-of-file, then

the built-in variable "`eof`" is set to "`yes`", but the unit is not closed unless a second attempt is made. (see "Detecting End-of-File" on page 98 for more details.)

The following is an example of "non-noisy" operation (so-called because "noise" is ignored, i. e., filtered out):

```
i1 = basopen("test","read")
integer i
real x, d(2,2)
i1 >> i >> x >> d
```

Let us suppose that input file "`test`" consists of the three lines:

```
c special input file
first = 2.56 , second = 13.51e-2
d = 1.2 2.3 3.4 4.5
```

Then after the execution of the above sequence of instructions, i will be 2 (the real value 2.56 having been coerced to integer), x will be .1351, d(1,1) will = 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.

To understand noisy mode operation (`noisy = yes`, i. e., "noise" is no longer ignored), it is first necessary to understand how the stream input *parser* interprets the incoming text. The parser divides the input stream into what we shall call "tokens", based upon the principle that it will build the longest legal token possible at each step. These tokens are as follows:

names begin with '%', '$', or a lower-case letter and may consist of zero or more additional '%', '$', '_', digits, or letters of either case.

group names begin with a capital letter and may consist of zero or more additional '%', '$', '_', digits, or letters of either case.

integers an optional sign followed by one or more contiguous digits.

reals an optional sign followed by one or more contiguous digits either containing a decimal point, or followed by 'e' or 'E' followed by an integer, or both.

doubles an optional sign followed by one or more contiguous digits either containing a decimal point, or followed by 'd' or 'D' followed by an integer, or both. Note that a double with a three digit exponent that has been written out by the stream output operator will not contain the 'd' or 'D' in its representation, so that only the first part of the number will be accepted.

strings contiguous non-delimiters which are not one of the previous five types of token.

Tokens are separated from one another by delimiters (currently spaces and commas). However, sometimes delimiters are not needed to separate tokens; e. g., "123abc" will be recognized as

"123" followed by "abc"; "abc.123" will be split into "abc" and ".123"; but "abc123" is a single token. For a more complicated example, "Joseph_Q._Jones" is three tokens, namely "Joseph_Q," "._" and "Jones."

In non-noisy mode, as noted previously, all tokens are ignored except for integers, reals, and doubles. In noisy mode, however, all tokens are significant, and there must be a variable in the input stream corresponding to each token. Furthermore, those variables corresponding to non-numeric tokens must be of character type or else chameleons. Use built-in function `type` to determine what has been read into the chameleon.

Consider, for example, the file "test" used in the preceding example. The following code will give i the value 2 and x the value .1351, as before, but will in addition assign to `name1` the string "`first`" and to `name2` the string "`second`":

```
i1 = basopen("test", "read")
integer i
real x
character*12 name1, name2
i1 >> $a >> $a >> $a >> $a #skip tokens on first line
i1 >> name1 >> $a >> i >> name2 >> $a >> x
```

There are a number of important points to note from this example:

1. `$a`, a chameleon, is used as a "sink" to receive unwanted portions of the input. Each unwanted token must be read to `$a`; it takes four such assignments, for instance, to discard the first line.
2. Note that blanks and commas *separate* tokens but are not themselves tokens. Thus the "... `i >> name2`..." in the second line of stream input automatically reads over the blanks and comma separating "2.56" from "second."
3. The value of `$a` after all of this is "=".
4. The first line of stream input could be replaced by i1 >> return
5. See section "Skipping Input Data" on page 99 for details.

We will now show three equivalent ways of retrieving input from a sample file called "data". "data" is a very short file consisting of six integer values, as shown below. Assume that i, j, k, l, m, n, and o are integers. "data" looks like this:

```
21 453  1
56,34   98765454
```

Method 1:

```
i=basopen("data", "read")
i >> j >> k >> l >> m >> n >> o
```

Method 2:

```
i=basopen("data", "read")
i >> j >> k >> l
i >> m >> n >> o
```

Method 3:

```
i=basopen("data", "read")
i >> j >> k
i >> l >> m
i >> n >> o
```

Each of the above methods opens the file “data” and reads the contents of “data” into the variables j, k, l, m, n, and o.

The user is allowed to have more than one input file open at once, up to a maximum (currently) of five, not including the terminal. If Basis is in the middle of an input line in one file when the user asks it to read input from a different file, it will keep its place in the unfinished line and resume from there if subsequently requested again to read from that file. If an error occurs in the read (either because of an incorrect assignment such as number to character, or because of an input error), then all open files will be closed.

23.3.1 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 which contains the value `no` if the last read attempt was successful, and `yes` if the last read attempt was unsuccessful. The user should test if `eof` is `yes` when performing input, so as not to attempt to read past the end of a file.

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.

Suppose, for example, we have an input file called “data”. Assume there are a variable number of inputs on each line, and an unknown number of lines. Once again i and j are integers. The user may read the input file as follows:

```
integer i,j
i=basopen("data","read")
i >> j                  # read first value
```

```
while (eof <> yes)       # if last read was successful
     call dostuff(j)     # process the value
     i >> j              # get next value
endwhile
```

Remember that `eof` indicates whether the last read was successful, and if the last read was not successful, j will retain its last value. Note also that `eof` is set by *any* unsuccessful read; if the read failed because of some kind of error, then the file will be closed. However, it will still be open if an actual `eof` was detected, and it is the user's responsibility in this case to detect the `eof` and close the file.

A couple of further words to the wise are in order. If `eof` becomes `yes` during a read operation involving several variables, even in the middle of a loop or if there are further variables to be read before `eof` is next tested, no error will result, and the subsequent variables simply will not be read. Finally, there is only one `eof` variable. If you happen to be doing alternate input from two or more different files, then `eof` could be set to `yes` by one file, and then reset to `no` by reading from the next. Thus one must be careful to test for `eof` before switching files.

23.3.2 Skipping Input Data

Basis provides a mechanism that allows users to skip certain portions of an input file. The word "return", used in an input command, tells Basis to ignore the remainder of the current input line, and to retrieve the next input from the following line.

As an example, consider the input file "junk" shown below. It is a file of integers:

```
23          45   56
98          76   54
12          34   78
89          21   43
67          90   87
```

Suppose the user only wants to read the first inputs on the second, fourth and fifth lines. This could be done as follows:

```
integer i,j,k,l
i=basopen("junk","read")
i >> return >> j >> return >> return
i >> k >> return >> l
```

The use of "return" depends upon where the parser is in the input line, and on the contents of the unread portion of the line. If there are non-null tokens yet to be read from the line, then a "return" causes parsing to skip to the start of the very next line. However, if the parser has fetched the last token in the line, there may be no characters at all left in the line, or the line might still

have characters on it which are only delimiters (and thus, possibly, invisible). In either case the "return" causes the parser to skip the next line and resume at the beginning of the second line following. This feature makes it unnecessary for the user to have to know whether input lines are blank-terminated before deciding how many "returns" to use to skip subsequent lines. Consider the following code (applied to the same file "junk"):

```
integer u,i,j,k,l
u = basopen("junk","read")
u >> i >> j >> k
u >> return >> l
```

After this sequence of instructions, i, j, k, and l will have the values 23, 45, 56, and 12, respectively. The "return" caused the second line to be skipped, even though the parser may still have been positioned before the end of the first line (because of the presence of blanks at the end of the line).

A user reading input from two or more files can use "return" as above to position the parser in the files. Basis always remembers its position in each of the opened stream input files. Thus interleaved "reads" and "returns" addressed to different files will always work properly.

23.4 The Output Operator <<

The user may direct Basis output to a terminal, to a plot, or to a file. For output to the terminal or a plot, invoke the output command as described below. For output to a file, first open the file using the function `basopen`, as described above.

We will now discuss the default Basis output command. The form of this command differs slightly depending on whether the output is being sent to a terminal, to a plot, or to an output file. The three forms of the Basis output command are:

```
<< output1 << output2...           #  output to a terminal
plot << output1 << output2...      #  output to a plot
unit << output1 << output2...      #  output to a file
```

where:

`output1, etc.` are the outputs. These may be integers, reals, doubles, or character strings. They can be scalars or arrays. Character arrays will presently only work if they do not contain any imbedded blanks.

`unit` is the unit specifier of the file to which the output is being sent (the result of the call to function `basopen`).

By default, each use of the output command produces one or more lines of output. If there is more output specified in the output command than will fit on one line, Basis will continue the

output onto extra lines. The exceptions to this are single strings that are longer than the maximum output line length of 80, and output commands using carriage control (see section on CARRIAGE CONTROL, below). If a string is longer than 80 characters, the first 80 characters of the string will be sent to the output unit. The remainder is discarded.

No spacing between outputs is provided by Basis. It must either be done explicitly or by use of the function format (see ”The Format Function” on page 102.) Here are some examples of `<<` output:

```
<< "This sends output to a terminal."
plot   << "Or to a plot."
x      << "Or even a file that has been opened."
ounit  << "i=" << i << "  " << "r=" << r
```

23.4.1 Carriage Control

Basis automatically provides a carriage return for each output command. Additional carriage returns may be inserted by the use of the word return in the output command. `return` may appear anywhere in the output command, and may appear as many times as the user wishes. For example:

```
<< return
<< x << return << y << return
```

Note that when `return` appears as the final output, the result is actually two carriage returns since one is still supplied automatically by Basis. A switch is available to suppress the automatic carriage return of the output command. By default, `autocr` is set to yes. The user may stop the automatic carriage return by setting `autocr` to no. Output is then buffered until a return is specified or the line buffer is exceeded, at which time the line is output. To stop the suppression of the carriage return, reset `autocr` to yes. For example:

```
autocr = no
i=4
<< "i=" << i << return
<< "j="
j=i+1
<< j << return
autocr = yes
<< "DONE"
```

produces the following output:

```
i=4
j=5
DONE
```

The user must exercise caution when output is being sent to more than one unit and the automatic carriage return is off. If the buffer is not empty when output is sent to a different unit, the buffered output may be sent to the wrong unit. Using RETURN at the end of an output command before sending output to a different unit will ensure that the buffer is cleared.

23.5 The Format Function

The user can format Basis output by using the function `format` which converts a numerical value to a character string. This string can then be used as an output in output commands, plot labels, etc. `format` needs two or four arguments depending on what type of number is being converted. Variations of the format function call are illustrated by this statement:

```
<< "iquad = " << format(iquad,0) << ", pi = " << \
  format(pi,0,5,1) << ", deficit > " << format(2e11,0,1,0)
```

which prints `iquad = -1234, pi = 3.14159, deficit > 2.0e+11`

23.5.1 Formatting Integers

`format` requires two arguments to convert an integer to a string variable. It needs the integer being converted, and the length (or field width) of the resultant string.

The general form is:

```
str = format(ival, fw)
```

where:

`str` is the character string returned by format. Ival is right-justified within str and str is blank-filled to the left.

`ival` is the integer being converted to a string.

`fw` controls the field width. If > 0, fw is the length of str. If fw = 0, str is just the length needed, without blanks. The field width must also be of type integer, and must be not be greater than the maximum length of an output line (132).

The maximum number of digits which may be converted using format is 14. If the user attempts to convert more than 14 digits, the resultant string will have an “r” in the right-most position. Likewise, Basis places an asterisk (`*`) in the right-most position if the field width is specified to be too small to hold the value being converted. Here are a few examples of correct and incorrect calls to format when the user wishes to convert an integer to a string. The resultant strings are also shown.

CALL TO format	RESULTANT STRING
str=format(784,0)	’784’
str=format(-456,0)	’-456’
str=format(784,6)	’ 784’
str=format(4.3,5)	ERROR: first argument is real
str=format(6,7.2)	ERROR: field width is real
str=format(78,567)	ERROR: field width too large
str=format(786,2)	`' *'` – field width too small
str=format(-456,3)	`' *'` – field width too small
str=format(9876543210987654,20)	`' r'` – too many digits

23.5.2 Formatting Reals and Doubles

`format` requires four arguments to convert a real value to a string. The user must provide the real number being converted, the length (or field width) of the resultant string, the number of digits to appear after the decimal point, and the form of the resultant string.

The general calling form is:

```
str = format(rval, fw, nd, ts)
```

where:

`str` is the character string returned by format. rval is right-justified within str, and str is blank-filled to the left.

`rval` is the real number being converted to a string.

`fw` controls the field width. If > 0, fw is the length of str. If fw = 0, str is just the length needed, without blanks. The field width must also be of type integer, and must be not be greater than the maximum length of an output line (132).

`nd` is the number of decimal places desired in str.

`ts` is the specification of the format of str. ts may be 0 to indicate D or E-format (5.467E+02 5.467D+02 or) or 1 to indicate F-format (546.7).

Different restrictions apply to the input parameters depending on whether the user wants E (D)-formatted output or F-formatted output. These restrictions are discussed next.

23.5.3 E (D)-format Restrictions

To obtain output in E (or D)-format, ts must be 0. The maximum allowed value for the field width fw is 32. If fw is zero, str will be just the length needed, without blanks. If fw is nonzero, fw

must be at least seven and the difference between fw and nd must not be less than seven. (This is because three places are required for the sign, leading digit, and decimal point, and four more for the exponent.) Otherwise, Basis places asterisks (*) in the string. Note that since four characters are always allotted for the exponent, in the case of doubles with a three digit exponent, the D is not printed. Such numbers can not be read correctly by the unformatted string input operator.

Below are some examples and results of calls to format on a workstation when D-format is the desired result.

CALL TO format FOR D-format	**RESULTANT STRING**
str=format(-450.67,14,4,0)	`' -4.5067D+02'`
str=format(-450.67,0,4,0)	`'-4.5067D+02'`
str=format(5.674,8,1,0)	`' 5.7D+00'`
str=format(1.23D123	`' 1.230+123' #No D`
str=format(6,15,1,0)	FORMAT:conversion of integer to string requires exactly two arguments
str=format(4.5,15,1,3)	FORMAT:type specification must be 0 , 1, or 2
str=format(4.5,3,1,0)	`'***'` #field width too small
str=format(4.5,8,3,0)	`'********'` # fw - nd < 7

(Note that on work stations, real literals default to double.)

23.5.4 F-format Restrictions

To obtain output in F-format, ts must be 1. The maximum number of digits which Basis returns is 32. If the field width fw is zero, str is just the length needed, without blanks. If fw is nonzero, fw must be at least 3 and the difference between fw and nd must not be less than 3. Otherwise, Basis places asterisks (*) in the string. If the value being converted is too large to fit in the specified field width, an "r" is placed in the rightmost position of the string. Below are some examples and results of calls to format when F-format is the desired result.

CALL TO format FOR F-format	**RESULTANT STRING**
str=format(-72.4,7,1,1)	`' -72.4'`
str=format(-72.4,0,1,1)	`'-72.4'`
str=format(7654.32145,14,3,1)	`' 7654.321'`
str=format(4.5,2,1,1)	`'**'` # field width too small
str=format(-4.654,5,3,1)	`'*****'` # fw-nd < 3

23.6 Closing File

If a user wishes to close a file, s/he may call the subroutine `basclose`. The form of this call is:

```
call basclose(unit)
```

where `unit` is the unit specifier of the file being closed. It is only necessary for the user to explicitly close a file using `basclose` if the file is currently open as an input or output file, and the user wishes to read that file starting from the beginning. If the user does not want to read an input file more than once, and does not wish to read an output file that has just been created using Basis output commands, then no calls to `basclose` are required.

CHAPTER

TWENTYFOUR

The Macro Facility

Basis has two types of macro definitions. The DEFINE statement is a small abbreviation facility whereas the MDEF-MEND statement is a full fledge macro facility. For either type of macro, a name can be defined as some body of text. Later, when that name is encountered as a token in the input, the body of text is substituted for it and rescanned for tokens. This means that substitution will NOT take place if the word name is inside a quoted string, occurs as part of another name, etc. Another way to keep a macro name from being expanded is to enclose the name within protection brackets { }'s.

24.1 Protection Brackets

The curly brackets { }'s will protect any macro name from being expanded. These brackets can also be used to protect the delimiters in macro calls and COMMAND statements. Protected delimiters will be treated as text and not as delimiters. However, any delimiters not in macro calls or COMMAND statments (such as the commas in a function call) can not be protected.

EXAMPLE:

```
integer x=5
DEFINE x 3
DEFINE title abc
MDEF name = fgh MEND
x
{x}                          ## protect a macro; don't expand it
g command_s {title name}     ## protect macros title and name
g command_s title{ }name     ## protect delimiting blanks in
                             ##   COMMAND statement
MDEF macargs() = some body MEND
macarg({a,b}, c)             ## protect delimiting comma in
                             ##         macro call
```

The results of the above examples will be to

1. print 3 # macro x is expanded to 3
2. print 5 # {x} references the integer x, not the macro
3. call function g with argument “title name”
4. call function g with argument “abc fgh”
5. call macro macarg with two arguments: a,b and c

24.2 DEFINE Statement

```
DEFINE name text
```

defines name to be an abbreviation for the text following up to the end of the line. It should be noted that a semicolon does NOT terminate the DEFINE definition. Rather it is included as part of the definition. This allows you to enter a semicolon delimited statement list in one DEFINE statement.

EXAMPLE

```
DEFINE X y;z
X
```

The above macro will cause both y and z to be printed.

Quotation marks around the definition of the macro are not required, but are allowed. If you wish the definition to be an actual string containing quotation marks, then you would need to double up the quotation marks. Thus the following two lines are equivalent.

```
DEFINE mymacro PLOT  y,x
DEFINE mymacro PLOT "y,x"
```

The following example could help you balance your checkbook:

```
DEFINE check "$-"
DEFINE deposit "$+"
355.66 #opening balance
deposit 433.44
check 55.22
check 12.98
```

is equivalent to the statements

```
355.66
$+433.44
$-55.22
$-12.98
```

which prints out the successive balances desired.

24.3 MDEF - MEND Statement

```
MDEF name = definition MEND
MDEF name () = definition MEND
```

By using the MDEF-MEND statement, you can define macros which allow arguments (up to nine arguments) and can have multiple line definitions. The first form of the macro (the one without the parentheses) is for macro which will never have a parenthesized argument list. The second form must be used if you ever wish to give the macro any arguments. Note: in the MDEF definition, the ()'s must not contain any argument names.

The words MDEF, the macro name, the ()'s if present, and the equals sign (=) must all appear on the same line. The rest of the definition can be spread over as many lines as you like. For example:

```
mdef mymacro =
   y
   plot y,x
mend
```

To reference a macro argument, use the notation $n where n, a digit from 1 to 9, is the number of the argument you wish to reference. If an argument is not present then the value of the argument is a 0 length string.

Besides the $n notation, there are two other macro argument notations: `$*` and `$-`. The notation `$*` refers to the entire argument list—separated by commas, but without parentheses—that the macro was called with. The notation `$-` refers to the entire argument list minus the first argument.

EXAMPLES:

```
mdef addargs() = integer $1 = $2+$3 mend
mdef allargs() = g($*) mend
mdef lessargs() = g($-) mend
mdef someargs() = $1;$2 mend
mdef noargs    = plot y,x mend
                         ## note macro declared without ()'s
addargs(x,5,7)
allargs(arg1, arg2, arg3)
```

```
lessargs(arg1, arg2, arg3)
someargs(arg1, arg2)
someargs(arg1)
someargs
noargs(arg1,arg2)
noargs
```

The above examples will expand to

```
integer x = 5+7                    ## addargs(x,5,7)
g(arg1,arg2,arg3)                  ## allargs(arg1, arg2, arg3)
g(arg2,arg3)                       ## lessargs(arg1, arg2, arg3)
arg1;arg2                          ## someargs(arg1, arg2)
arg1;                              ## someargs(arg1)
;                                  ## someargs
plot y,x(arg1,arg2)                ## noargs(arg1,arg2)
plot y,x                           ## noargs
```

The expansions of the macros addargs, allargs, and lessargs are straightforward given the definitions of $n, $*, and $-. The expansions of someargs and noargs are a little more complicated.

The expansions of macro someargs shows you what happens when not all the referenced arguments are present. The first expansion of someargs is straightforward. The notation "$1" is replaced by the text "arg1" and "$2" is replaced by "arg2". In the next expansion no second argument is present. Thus "$2" is replaced by a null string, and the resulting body is "arg1;". In the next example no arguments are present. Thus both "$1" and "$2" are replaced by null strings, resulting in a body of ";".

The expansions of macro noargs show you what happens when a macro is declared without arguments. Remember that this macro was declared without parentheses ()'s. Thus a parenthesized list following the macro name is NOT ever considered part of the macro call. Therefore the word "noargs" is expanded to "plot y,x" and the words "noargs(xarg1,arg2)" is expanded to "plot y,x(arg1, arg2)", i.e. the word "noargs" is expanded and the words "(arg1, arg2)" are left as they were found.

24.4 IFELSE Statement

```
IFELSE (arg1, arg2) (arg3, arg4)
IFELSE (arg1, arg2) (arg3)
```

The IFELSE statement takes two argument lists. The first list contains the arguments of the if test. The second list contains the arguments of the if selection. The IFELSE macro is replaced by the text of one of the arguments in the if selection, depending upon the result of the if test.

The two arguments in the if test are first expanded of all macros and then the resulting text of each argument is compared against each other. If `arg1` is identical to `arg2`, then the IFELSE statement is replaced by the text of `arg3`. Otherwise the IFELSE statement is replaced by `arg4` if it is present. If `arg4` isn't present (and `arg1` and `arg2` aren't identical), then the IFELSE statement is replaced by a zero length string, i.e. it expands to nothing.

For example:

```
mdef Dim() = real $1 ifelse ($2, ) ( , ($2))  mend
Dim(x)
Dim(y,100)
```

The above example expands to

```
real x
real y (100)
```

Remember that if an argument is not present, the `$n` notation for that argument expands to nothing. Thus the if test `($2, )` of the above IFELSE statement will determine if the Dim macro was called with a second argument. If a second argument is present then the if test `($2, )` will be false, causing the IFELSE statement to expand to `($2)`. Otherwise the IFELSE statement will expand to nothing.

24.5 UNDEFINE Statement

```
UNDEFINE namelist
```

namelist is a blank and/or comma delimited list of names to be removed from the table of macro definitions. For each name in namelist, the UNDEFINE statement will remove the definition, regardless of whether the macro was originally defined with the DEFINE or MDEF-MEND statements. For worriers: when the words DEFINE, MDEF, or UNDEFINE are seen, macro expansion is turned off so that name is not expanded, thus giving us the chance to see the name so that we can re-DEFINE or UNDEFINE it!).

CHAPTER

TWENTYFIVE

Executing System Commands from the Parser

The user can execute any system or shell command easily from the parser. To do this simply enter as in unix:

! commandline

Example: To give a long list on the Sun of files ending in ’src’:

```
! ls -l *src
```

CHAPTER

TWENTYSIX

Timing

```
TIMER ON | OFF
```

TIMER ON starts a clock running. TIMER OFF prints out the timing statistics since the last TIMER ON. For something fancier, see “TIM: Interrupt Timing” on page **??** of *The Basis Package Library* document. Here is the *OFFICIAL* interface to the system timing routines:

The Fortlib routine timeused has a different number of arguments depending on your system: NLTSS, Sun, UNICOS, ... We hereby publish an *official* interface to the system timing routines:

subroutine ostime(cpu,io,sys,mem) real cpu, io, sys, mem

subroutine glbwrtim(iunit,cpu,io,sys,mem) integer iunit real cpu, io, sys, mem

The glbwrtim routine will print out and label correctly the quantities obtained by ostime. cpu and sys are guaranteed to be in seconds, and represent cpu and system time, respectively. On any system, io is some measure of the io effort, and mem is some measure of the amount of memory resource consumed. The numbers returned by ostime increase monotonically with time. A bigger number is more. That’s all we officially know.

CHAPTER

TWENTYSEVEN

Ending Basis

```
END
quit
quit(1)
```

END terminates the execution of the program. Currently, END must not occur inside a structured statement. The function quit has the same result but may be called from anywhere. If an argument is given for quit, it is used as the exit status.

CHAPTER

TWENTYEIGHT

Error Recovery

In an interpreted language, it is often possible to recover from errors. When an error occurs Basis does its best to help the user understand the nature of the problem. A user variable `debug` governs the error message that goes to the terminal and logfile. The variable `debug` can be assigned the values `yes` or `no`. The default value is no, and error messages will be brief. If `debug` = `yes`, a much more thorough error message goes to the terminal and logfile. If Basis was executing, information is given about the location where the error occurred; a complete trace of all the local variables and arguments to any functions is given, and some symbolic information about the error is given. Whether `debug` = `yes` or `no`, a file is produced that contains the debug information. The filename is first the contents of the variable `probname`, then a numerical extenstion, then the file type appended.

Here is an example. The call to function `boom` fails when it attempts to add two arrays that are not the same length. The error message, "Operands not compatible in size for +" is followed by a more detailed error analysis because `debug = yes`.

```
Basis> function boom(a,b)
Basis> chameleon temp
Basis> temp= (a+b)/2
Basis> return [temp,temp**2]
Basis> endf
Basis> debug=yes;boom([2,2],[2,3,4])
parcnfm: Operands not compatible in size for +
Writing traceback info to file problem.001
Returned to user input level.
```

The relevant contents of the file problem.001 are:

```
Here is the information I have on where you were:
   A call to boom containing
   the problem.
The error occurred in the assignment or append statement:
               temp = expression
The following lines contain clues(not facts) about the r. h. s.
```

```
b
+
a+b
temp
Parser’s action number = 115(ADD), program counter = 45.
Group: Locals_boom  Num Vars: 3
a(2)
 1:     -  2 2
b(3)
 1:     -  2 3 4
temp     =  0
```

If the parser function `errortrp("off")` has been called, no error recovery is attempted.

A compiled routine `kaboom(iflag)` can be called from the parser to force a return to the parser. If iflag is greater or equal to 0 Basis acts as if an error has taken place, and produces a trace file. If iflag is set negative, Basis returns to the parser without any error messages.

```
function subt(a,b)
if( a < b) then
     remark "Error: subt called with a < b"
     call kaboom(-1)
endif
return a-b
endf
```

In interpreting the information printed out when `debug` = `yes`, you should begin with the error message itself, examine the description of the nesting levels to find out where you were generally, and examine the symbols listed for some hints about the parts of the expressions involved in the error. As a last resort, the pc counter can be used in conjunction with the list command. The value of the pc counter is given in the `trace` file. Do a list on the function in which the error occurred, and when asked answer ‘y’ to the question about viewing the intermediate code. The listing which results shows the operations being performed and using the pc you should be able to pinpoint which instructions caused the error. For example, in the example above, boom reported that the error occurred at pc = 45.

```
Basis> list boom
boom(a,b)
    user-defined function
Minimum number of arguments:     2
Maximum number of arguments:     2
User-defined function, begins at absolute address      3967064
Function consists of    104 words of intermediate code.
 NAME                    TYPE
 boom                      varies
```

```
 a                          varies
 b                          varies
Dump intermediate code? (y|n)
y

   pc   opcode  stack     operation
    1:     16          Enter function, set up actual parameters.
    3:    421      7   REGULAR_SCOPE
    5:    404      9   CHAMELEON
    7:   -100     10   TOKEN = 'temp' (name).
   12:    412     10   SCALAR DECLARE
   14:    415     10   NO INITIAL VALUE
   16:    410     11   CREATE VARIABLE
   18:   -100      8   TOKEN = 'temp' (name).
   23:      1      8   ID->LHS
   25:     10      9   BEGIN RHS
   27:   -100     11   TOKEN = 'a' (name).
   32:      1     11   ID->LHS
   34:    136     11   <LHS>-><FACTOR>
   36:   -100     13   TOKEN = 'b' (name).
   41:      1     13   ID->LHS
   43:    136     13   <LHS>-><FACTOR>
   45:    115     13   ADD
   47:    130     12   MOVE-1
   49:    136     10   <LHS>-><FACTOR>
   51:   -101     12   TOKEN = '2' (integer constant). value = 2.
   56:     11     12   PUSH VALUE

   58:    121     12   DIVIDE
   60:      3     10   ASSIGN
   62:   -100     10   TOKEN = 'temp' (name).
   67:      1     10   ID->LHS
   69:    136     10   <LHS>-><FACTOR>
   71:      8     10   FETCH VARIABLE
   73:   -100     12   TOKEN = 'temp' (name).
   78:      1     12   ID->LHS
   80:    136     12   <LHS>-><FACTOR>
   82:   -101     14   TOKEN = '2' (integer constant). value = 2.
   87:     11     14   PUSH VALUE
   89:    126     14   POWER
   91:    101     12   EXPLIST
   93:    129     11   [EXPLIST]
   95:    136      9   <LHS>-><FACTOR>
   97:     19      9   FETCH COPY
   99:     17      9   RET
```

```
  101:     18      8   NULLRET
  103:     17      9   RET
**************************
```

CHAPTER

TWENTYNINE

Interrupting Basis

Basis can be usually interrupted by typing control-C. The terminal interactions of the operating system sometimes make it hard to do this if a lot of output is being displayed and several tries may be required. Basis checks for this message before each step of intermediate code and before most lines of output. The routine `ruthere` looks for the control-C message. If you are executing in a compiled routine the interrupt may not work unless the author has inserted "`call ruthere`" in the program at strategic points. Annoy your author until he or she does so. However, if the `ctl` package `run` command or its relatives are controlling the operation of a physics package, a check is made after each step and authors need not include `ruthere` calls in that case.

CHAPTER

THIRTY

List of Reserved Words

Basis reserved words are written in upper case throughout this manual for purposes of emphasis, but are recognized by Basis if they are entered entirely in lower case. These words cannot be used as identifiers:

```
BREAK, CALL, CHARACTER, COMMAND, COMPLEX, DEFINE, DO,
DOUBLE, ELSE, ELSEIF, END, ENDDO, ENDF, ENDFOR,
ENDIF, ENDWHILE, FOR, FORGET, FUNCTION, IF, IFELSE,
INDIRECT, INTEGER, LINLIN, LINLOG, LIST, LOGICAL,
LOGLIN, LOGLOG, MDEF, MEND, NEXT, OUTPUT, PLOT,
PLOTM, RANGE, READ, REAL, RETURN, THEN,
UNDEFINE, UNTIL, WHILE.
```

CHAPTER

THIRTYONE

List of Non-Alphanumeric Tokens

!	+	<	[
%	,	<=	]
&	-	<>	—
’	.	=	~
(	/	==	~=
)	/!	>	(space)
*	//	>=	(return)
*!	{	?	‘
**	}	@	:
;	^	—=	&=
+=	-=	*=	/=
**=			

Here is a list of the non-alphanumeric tokens used in basis. Any input character other than one of these, which is not alphanumeric, will cause an “illegal character in input” error (unless, of course, it is part of a comment).

CHAPTER

THIRTYTWO

List of Parser Variables

32.1 Variables

asgnchek controls whether or not the limiting string of a variable is used in determining the bounds of an array which is the target of an assignment statement. The default value is `yes`. Use `asgnchek=no` to allow storage to array elements outside the current `setlimit`'ed values.

autocr If = `yes`, then each output command, `<<`, will automatically supply a carriage return. If `autocr = no`, then no carriage return is automatically supplied by Basis. Output is buffered until either a RETURN is included in an output command or the buffer is exceeded. Default = `yes`.

autodyn when `autodyn = yes` any attempt to access a dynamic array will cause storage to be allocated for it if it does not already have it. There are two auxiliary variables which are used when this occurs: `autodynp` is an integer containing an amount of padding to be added to such arrays, and `autodyna`, if set to a non-blank string, will give that attribute to any array allocated in this way. These two variables default to 0 and blank, respectively. The default value of `autodyn` is `no`.

autohist Controls the handling of display statements. If it is set to 0, each displayed quantity is assigned to the variable `$` and then displayed. This is the default.If `autohist <> 0`, the displayed quantity will be assigned to one of the 26 variables `$a`, `$b`, ...,`$z` depending on `mod(autohist,26)`, with `$a` corresponding to `autohist = 1`, `$b` to `autohist = 2`, etc. Then `autohist` will be incremented. So, if you set `autohist = 1` to begin with, the results printed will be saved in `$a`, `$b`, etc., and after `$z` back to `$a`. Thus a "history buffer" of 26 previous results will be maintained.

autovar when `autovar = yes` undeclared and unsubscripted variables are automatically declared. Default is `no`.

compress if `yes`, compress on output repeated array elements; if `no`, list each element of an array separately. For more extensive control see "The Stream I/O Facility" on page 93.

cprompt can be set to any character string up to length 16 to change the basic prompt.

debug is set to `yes` or `no` to control the amount of detail in the error printout. `debug` = `yes` causes extensive printouts; `debug` = `no` does not, but the program executes much faster. The default value is `no`.

dec toggle the output to occur in decimal form. The default.

debuga used to control detailed debugging printouts

debugc If `yes`, print stack dump before and after each action.

echo is set to `yes` (1), `no` (0), or `logonly` (2) to control echoing of lines from input files. Default is `yes`. If `echo` is `logonly` then lines are echoed into the log but not to the terminal.

eof an integer which contains the value `no` if the last read attempt was successful, and `yes` if the last read attempt was unsuccessful.

fuzz number of digits after decimal point in prints. Default = 5. For more extensive control see "The Stream I/O Facility" on page 93.

hex toggle the output to occur in hexadecimal form. Decimal (`dec`) is the default.

coredump if `yes`, dump core when exiting if the exit status is not 0. Default value is `no`. An obsolete but still working alias for `coredump` is `keepdrop`.To disable the system's error recovery, type: call `errortrp`("off"). To restore error recovery type: call `errortrp`("on").Exit status is set non-zero when exiting because error recovery is off or if routine `quit` is called with a non-zero argument.

lcprompt should be set to the number of characters in `cprompt` that are to be used.

lsprompt should be set to the number of characters in `sprompt` that are to be used.

noisy If set to `no`, ignore all non-numeric tokens ("noise") in stream input. If set to `yes`, all tokens are significant and are to be assigned to a corresponding input stream variable. Default is `no`.

notty If `yes`, terminate the run after processing the macfiles. Otherwise go on to the Basis prompt. Default is `no`.

nskipr n If n $>$ 0 skip the first n records on the next file read. Basis automatically resets `nskipr` to 0.

oct toggle the output to occur in octal form. Decimal (`dec`) is the default.

padding Each call to `allot` or `change` allocates a certain number of elements. To this amount, `padding` extra elements will be added. The extra space is not used by Basis in any way.

sprompt can be set to any character string up to length 16 to change the secondary prompt.

switches An array of 100 real switches. Defaults to 0.

verbose If set to `yes`, print out all the system messages to the TTY and the log file. Default is `yes`.

32.2 Constants

blank is a 80-character variable full of blanks

false contains the logical constant .false.

off contains “off”; many packages expect “on” or “off” as the setting for devices or plot options.

on contains “on”; many packages expect “on” or “off” as the setting for devices or plot options.

no contains an integer 0; many packages expect `yes` or `no` as values for switches, such as `echo` above.

pi contains the real value of pi (3.14159...).

stdin contains the unit number for reading from the terminal.

stdout contains the unit number for writing to the terminal.

stdplot contains the unit number for writing to the plot file.

true contains the logical constant .true.

yes contains an integer 1; many packages expect `yes` or `no` as values for switches, such as `echo` above.

CHAPTER

THIRTYTHREE

List of Compiled Functions

This section describes functions that are callable from the Basis Language. These routines are all ordinary compiled Fortran and you can pass them arguments by value (the default) or by address (using the form `&x`). (see “Built-in Functions” on page 51.)

33.1 Working With Attributes

Each named entity (variables, functions, and macros) can have zero or more “attribute” words associated with it. These words are then available as keys to select names on which special functions called “attribute servers” can operate. Normally attributes are given to variables by program authors. Users may give or remove attributes using the function rtcattr:

```
call rtcattr("name","attributes")
```

Here attributes is a space delimited list of attribute words (up to 24 characters). A word can be prefixed with a minus sign to remove an attribute from a word.

The existence of an attribute can be tested with the function rtattr:

```
iflag = rtattr("name", "attribute")
```

`iflag` will be `TRUE` if the attribute exists and `FALSE` otherwise.

Writing attribute servers is explained in the document Writing Basis Programs: A Manual For Program Authors. See “Writing Attribute Services” on page **??**. There are two servers built in to Basis:

```
call attrlist(aexp,iunit)
call attredit(aexp,iunit)
```

Here `iunit` is a unit number, and `aexp` is a quoted string containing a logical attribute expression. A logical attribute expression is built up from attribute names, parentheses, and the operators

& (and), | (or), and ~ (not). Plus and minus can be used as synonyms for | and -. The routines respectively list, or edit the variables whose attributes satisfy `aexp`, to the unit iunit.

If a user wants to print the values of certain variables, for example, she might call `rtcattr("name","mylist")` for each variable name desired. Then `attredit("mylist",stdout)` will print all such variables to the terminal.

33.2 Help and News

Subroutines `help` and `news` supply information about the help package and the most recent changes, respectively.

33.3 Memory Management of Dynamic Arrays

The following routines are more thoroughly documented in Writing Basis Programs: A Manual For Program Authors. See “Writing Basis Packages” on page **??**.

`call allot("array",length)` allocates an array of length elements. The quotes around the name are required. If array is a multidimensional array, length is the length of the desired last dimension of array. The database manager knows the type and other dimensions of array. The package to which array belongs is determined by the current context. Each element would contain 2 words if array is complex.It is not an error if array has already been allocated space that has not been released by a call to basfree.

`call basfree("array")` Releases space for array previously obtained by a call to `allot`. The quotes around the name are required.

`call change("array",newlength)` changes the length of array to `newlength`. The quotes around the name are required. The comments above about multidimensional arrays apply here as well.

`call gallot("name",n)` allots all the dynamic arrays in group, name.

`call gchange("name",n)` changes the allocation of all the dynamic arrays in group, name.

`call gfree("name")` frees all the dynamic arrays in group, name.

33.4 Opening and Closing Files

`integer basopen`

iunit = basopen(name, access) This routine is used for opening input files and for creating output files.If access is “r”, opens file `name`, returning the unit number to use in subsequent operations. If the file is not present, it is searched for (using the list in variable `path`, which can be added to with the variable codefile in config, or by the routine pathadd). Error recovery is invoked if the file cannot be found at all.If access is “i”, `basopen` returns OK or ERR (0 or -1) to indicate whether or not the file can be opened in “r” mode.If access is “w”, the file is created, returning the unit number to use in subsequent operations. Error recover is invoked if the file cannot be created.Any file opened with `basopen` will be CLOSED whenever error recover takes place.

call outfile(&j,comment) opens a text output file and places the value of the unit specifier into the integer variable `j`. Note that since `j` is an output quantity it must be passed by address. `comment` is a character string that is used to comment the file. When Basis terminates, if `verbose` is yes, the files created by `outfile` will be listed along with the comments supplied. Basis generates the name of the file from a combination of the value of `probname`, and a counter. Files created with outfile are NOT closed when an error occurs. If comment = “*temporary*”, the file is deleted when the program terminates. `basclose (unit)` should be used to close files opened with outfile.

call basnxtsq(f,g) (Fortran)

g = basnxtsq(f) (Basis) given a filename f sets g to the next name in the sequence when called from Fortran, or returns the next name in the sequence when called from Basis. In Fortran, f and g may be the same variable. Some of the sequence types handled by basnxtsq, using an algorithm of Dave Munro’s, are: prob01fa prob01fb prob01fc prob01fd ... prob02q00 prob02q01 prob02q02 prob02q03 ... prob02q43 ... prob03q00.pdb prob03q01.pdb prob03q02.pdb prob03q03.pdb ... prob03q00.cdf prob03q01.cdf prob03q02.cdf prob03q03.cdf ...

33.5 Executing User Functions

You can execute a user function `f` by

call execuser("f") The function `f` must have no arguments and cannot return a value. This function is usually called from compiled code.

33.6 Adding Comments to Variables and Functions

call comment("name","comm") adds a comment to any variable or function in the database including those defined by the user.

33.7 Checking for the Existence of Variables and Functions

You can check whether a variable or function has been defined by typing:

```
if(exists("name"))
```

33.8 Flushing the LogFile

`call flushlog` flushes the log file. This can be useful after an error recovery if some vital information has scrolled off your screen.

33.9 Using the Switches Array

You can set switches(i) = x if you type:

```
call swset(i,x)
```

You can get the value of switches(i) by invoking the function switch:

```
switch(i)
```

33.10 Protecting User-Defined Variables and Functions

You can protect user-defined functions and variables made up to this point from future FORGETs by typing:

```
call protect
```

33.11 Setting Variable Dimension Limits

`setlimit("name", "( dimension )" )` allows you to restrict the portion of an array that will be used. The function setlimit can be called from user or compiled code. Setlimit uses the usual search to determine the meaning of name. The parentheses in the second argument are required. The restrictions on dimension are the same as for regular dimensioning strings: the contents of the string must consist of constants, operators, and names which can be evaluated. The allowed operators are `+, -, *, /`. In evaluating names in this string, the database for name is searched first; only if this fails is the usual search made.Subsequent accesses to this array cause a reevaluation of the limiting string so that changing variables

which appear in the limiting string will change the amount of the variable used.The limiting string must define the correct number of dimensions and the values for the limits must be within the storage dimension values. Exception: the upper limit for any dimension may be set to one less than the lower index of the array, thus declaring that no part of the array is currently in use.

`setlast("name", n)`

`rtadddim("name")` The routine `setlast("name", n)` limits the LAST dimension (only) of the variable name to a high subscript of n. If n is greater than the current highest (unlimited) last subscript of name, then an attempt is made to expand storage so that n will be a legal subscript. The size of the last subscript is increased to the maximum of n and its current value times 1.5 with a minimum of 16. `setlast` can be used on static arrays as long as no attempt is made to exceed the actual storage available.The routine `rtadddim("name")` adds a dimension to the variable name, which is sometimes useful in conjunction with setlast and the append statement.

33.12 Specifying Assignment Actions

For each variable, the user may specify a string containing Basis language statements called its assignment-action string. This string will be parsed and executed after each assignment statement in which the corresponding variable name appears on the left-hand side of the assignment statement. To set the string do:

call `setact("name", "action" )` Any subsequent assignment statement which changes the variable `name` will cause the action string to be parsed and executed. Restrictions are: the action string must consist of complete statements (e.g., compound statements like do loops are fine but must be complete); the action string must be 72 characters or less; the action should not induce an infinite recursion (such as `real w;setaction("w","if(w<0) w=-1.")`). Examples: `real x; call setact("x","x")` will cause the value of x to be printed whenever it is changed with an assignment statement. (Useful for debugging!). If y is a parameter which is supposed to contain a value between 0. and 1., an author might do something like: `call setact("y", "if(~ (y ? [0.,1.])) then; remark ""Bad y"";kaboom(0);endif")` to prevent the user from assigning a bad value.Since the action may include a call to any function, the restriction on the length of the string can be easily finessed.

33.13 Redefining Array Shapes

`setshape("name", "( dimension )" )` allows you to reset the dimension statement of a variable. It does not change the storage allocated, merely the perceived shape of the array. If the number of items in the new dimension does not equal the current number of

items an error is issued. See `setlimit` above. The function `setshape` can be called from user or compiled code. `setshape` uses the usual search to determine the meaning of name. The parentheses in the second argument are required. The restrictions on dimension are the same as for regular dimensioning strings: the contents of the string must consist of constants, operators, and names which can be evaluated. The allowed operators are `+`, `-`, `*`, `/`. In evaluating names in this string, the database for name is searched first; only if this fails is the usual search made.

`useshape("name")` evaluates the existing symbolic dimensions and assigns the shape to the variable. This is useful when a variable already has a symbolic shape that is assigned by `rtvare` but the memory is allocated by the client code. This call has the effect of causing Basis to use the current shape of the memory. The function `useshape` can be called from user or compiled code. `useshape` uses the usual search to determine the meaning of name.

33.14 Functions With Variable Numbers of Arguments

If the parameter list of a user-defined or compiled function contains a semicolon at the beginning or in place of one of the normal commas, the arguments which follow the semicolon are optional. The function can be called with none, some, or all of its optional arguments.

Additionally, you can use the function `setmnarg` to declare optional arguments for both user and compiled functions.

`setmnarg("name",n)` sets the minimum number of arguments to the function `name` to be `n`. The function must be a user or compiled function that has at least `n` arguments.

When a user calls the function without all of its arguments, what happens depends on whether the function is user-defined or compiled.

For user-defined functions, the local variables corresponding to the arguments which were not supplied are simply not created, which will cause an error if an attempt is made to access that name *unless* a variable with the same name as the formal parameter exists in the search path. This can be used to set default arguments. For example, if I have a physics variable named gamma which is the usual third argument to a function f, then I can write f as follows:

```
function f(a,b;gamma)
.....
endf
```

after which f(1.,3.) results in the physics variable gamma being used as the third argument, while f(1.,3.,5.) uses 5. as the third argument. This works because when the call f(1.,3.) occurs, no variable named gamma gets created in the local variables for f, and so references to gamma in f become references to the gamma in the search stack.

A macro `default` is built in to Basis which makes it easy to supply default values locally. The usage is:

```
default(name) = value
```

If the `= value` portion is omitted, name will be created as an integer scalar with value 0 if the argument name is not supplied. If the `= value` portion is given, and the argument name is not supplied, name is created as a chameleon variable and value is assigned to it.

For compiled functions, Basis will fill in the missing arguments by passing one of the following values, depending on the type of the argument:

type	default value
integer	DEFAULT
real	float(DEFAULT)
complex	(float(DEFAULT), float(DEFAULT))
logical	FALSE
character	rone blank

The constant DEFAULT is supplied by MPPL for authors to use to decide if an optional argument was omitted or not.

33.15 Creating Pauses

`call paws` which causes Basis to pause and request a carriage return to continue. If the user sends any other message an error exit is taken through `kaboom`.

33.16 Returning to the Parser

You can force a return to the prompt with the statement:

```
call kaboom(iflag)
```

If `iflag <> 0 & debug=yes`, this can create a long very useful printout.

33.17 Recursive Parsing

You can compile and execute a statement in the Basis Language with the subroutines `parsestr`, `parselng`, and `parse`:

```
character*(n) s   # n a number <=500
character t(m)
character*(n) u
```

```
call parsestr(s)  #or,
call parselng(t,m) #or,
call parse(u,n)
```

Restriction: the string to be parsed cannot contain a READ statement.

These routines can be called from anywhere within the Basis environment: from the interpreter, from a compiled routine (for instance, in a physics package), from a built-in routine, or even through some"hook" in a graphics library. It may be called recursively to any depth; for instance, it may be asked to parse a string which itself contains a parsestr call. Each time it is called, it saves sensitive portions of its environment on a stack, thus making this flexibility possible.

Thus,

```
call parsestr("global real x = 3.")
```

will execute the statement "global real x=3.", creating a real variable x whose value is 3.

Any variables created without the "global" scope are created in the stack frame of the parsestr function itself, and will therefore disappear on return. Any user-defined functions declared will be defined after return. If the string does not represent a series of complete statements those statements not yet completed are discarded without execution. If a syntax error or semantic error occurs, error recovery occurs as usual and one is returned to the bottom level parser.

The maximum length of strings is a system-dependent limit (about 500 on Crays).

The routine `parselng` allows you to exceed this by using a character array. And the routine `parse` allows you to pass an array of strings which you wish treated as a sequence of lines; `parse` will insert semicolons between the array elements and pass the result to `parselng`.

33.18 RANF and Its Supporting Routines

Ranf is a 48 bit multiplicative congruential method RNG which produces 64 bit floating point numbers in the open interval (0,1). More precisely, it produces a sequence of uniform variates U_i based on the following formulas:

$$U_i = \frac{S_i}{2^{48}} \tag{33.1}$$

$$S_{i+1} = aS_i \bmod 2^{48} \tag{33.2}$$

where the (integer) multiplier a = 0x2875a2e7b175 [1] and the (integer) default seed S_0 = 0x948253fc9cd1. Note that the minimum value for U_i is $1/2^{48} \cong 10^{-15}$ and the maximum value is $1 - 1/2^{48}$. [2]

[1] The multiplier's inverse is a' = 0x5ceeb894d6dd, with $aa' \equiv 1 \bmod 2^{48}$.

[2] If stored in an IEEE 754 Standard single precision (32 bit) floating point format, the minimum is distinct from 0; however, the maximum (and many other values near it) are not distinct from 1.

On the Cray, *ranf* is loaded from the *Mathlib* library. The workstation version is based on the *drand48* suite of library functions. Although these two libraries implement the same basic arithmetic, there is a subtle difference in that *drand48* computes the next seed and returns that value divided by 2^{48}, whereas *ranf* saves the old seed, computes the next, and then returns the old seed divided by 2^{48}. The result is that the sequence from *drand48* is "one ahead" of that from *ranf*. This problem may be solved by decrementing or incrementing the seed by one as part of setting or retrieving it, respectively, and this logic is built into the workstation versions of *setranf* and *getranf* routines below. Thus sequences generated by the Basis *ranf* are identical on Cray or workstation, and seed values may be carried between the two architectures without a break in the sequence.

The examples below are written as they would appear in source to be preprocessed by *Mppl*. In Fortran terms, *ranf* will return *double precision* on a 32 bit workstation, and *real* on a 64 bit architecture.

33.18.1 Ranf

```
real(Size8) ranf,x
x = ranf(0)
```

Ranf may also be called from the Basis interpreter as a built-in function.

33.18.2 Getranf

```
integer iseed48(2)
call getranf(iseed48)
```

Getranf reads the current 48 bit seed, placing the lower 32 bits in *iseed48(1)* and the upper 16 bits in *iseed48(2)*. It may be called from the Basis interpreter, but be careful to pass its argument "by reference" in that case: *call getranf(&iseed48)*.

33.18.3 Setranf

```
integer iseed48(2)
call setranf(iseed48)
```

Setranf restores a 48 bit seed (presumably stored in *iseed48* by an earlier call to *getranf*). If both elements of the array are zero, the default seed is reset. *Setranf* may be called from the Basis interpreter as a compiled function.

33.18.4 Seedranf

```
integer iseed
call seedranf(iseed)
```

The semantics of *seedranf* are similar to Cray Mathlib's *ranset*, with some restrictions. If *iseed=0*, the default seed value is restored. Otherwise, the given value (or the next odd integer if *iseed* is even) is set as the current seed. If you're setting an arbitrary seed, be aware that integers on the workstation are usually limited to 32 bits, and that the upper 16 bits of the 48 bit seed are set to zero by this call. Thus, the first value returned by *ranf* will be quite small (

).*Seedranf* is provided primarily as a convenient way to reset the default seed - e.g., "*call seedranf(0)*" This function is also available within the Basis interpreter.

33.18.5 Mixranf

```
integer iseed, iseed48(2)
call mixranf(iseed,iseed48)
```

Mixranf provides functionality similar to Mathlib's *rnfmix*: If *iseed*¡0, the default seed is set. If *iseed*=0, then a "random" seed is created from the system clock plus 10 calls on *ranf*. If *iseed*>0, then the value is set directly as per *seedranf*, with similar wordsize restrictions. *Mixranf* can be called from the Basis interpreter.

33.19 Manipulating the External Environment

These functions do some (simple) things that are otherwise hard to accomplish inside Basis programs. Here's what's currently available:

33.19.1 basisexe()

NAME

`basisexe()` - execute a shell command

SYNOPSIS

```
integer status
status = basisexe("ls ~/wrk")
```

DESCRIPTION

The `basisexe` function may be executed either from FORTRAN code or directly from the command line at runtime (although use of the shell escape '!' requires less typing). The argument should be a quoted string containing a legal shell command (or commands, separated by semicolons). This function returns the status code of the command (normally zero unless there was some error).

33.19.2 cd, chdir()

NAME

`cd, chdir()` - change working directory

SYNOPSIS

```
cd /foo/bar
logical chdir("/foo/bar")
```

DESCRIPTION

The `cd` command works almost exactly like its counterpart in the UNIX shells. It accepts the standard shorthand meaning for the tilde "~" character. `cd` with no arguments changes to your `HOME` directory. For use in scripts, the `chdir()` function is provided. It returns logical true or false depending on whether the command succeeded or failed. Typical use:

```
    if(chdir("/foo/bar")) then
    remark "it worked!"
else
    remark "try something else"
endif
```

33.19.3 setenv, getenv

NAME

`setenv, getenv()` - set or read environment variables

SYNOPSIS

```
setenv foo bar
character *64 homedir = getenv("HOME")
```

DESCRIPTION

`setenv` works just like its counterpart in the C shell, and is occasionally convenient for resetting, say, `NCARG_ROOT` from within a running Basis program. The `getenv()` function (which formally returns "character *(500)") is provided for symmetry, and to make it easier to set a Basis variable to the value of a given environment variable. For example,

```
chdir(getenv("PWD"))
```

will usually set the working directory to the value it had when you started a Basis session.

33.19.4 diskspace

NAME

`diskspace()` - find the remaining free space in your file store

SYNOPSIS

```
real xxx = diskspace("/foo/bar")
```

DESCRIPTION

The `diskspace()` function returns the number of megabytes of space available to you in the filesystem containing its pathname argument. If no argument is given, the current working directory is assumed. Diskspace attempts first to determine if you have a quota assigned on the filesystem in question. If you do, it returns the amount of free space available before you hit your hard limit. If you don't have a quota, it simply returns the available free space on the disk, just like the "df" command. It returns -1 on error. Typical usage:

```
    cd ~/wrk
if(diskspace() < 20.0) then # Quit if less than 20MB free
    fin
else
    remark "Running another cycle"
endif
```

INDEX

Symbols

A

B

C

D

E

F

S

T

U

V

W

Y

Z