MacAnova Version 4.07

This file consists of Chapter 9 of **MacAnova User's Guide** by Gary W. Oehlert and Christopher Bingham, issued as Technical Report Number 617, School of Statistics, University of Minnesota, revised August 1998, describing Version 4.07 of MacAnova.

Fonts used in this manual are Palatino, Courier, and Symbol.

For information concerning MacAnova, write University of Minnesota, Department of Applied Statistics, 352 Classroom Office Building, 1994 Buford Avenue, St. Paul, MN 55108-6042.

9. Programming MacAnova

9.1 **Working with structures** There are several functions useful for working with structures. We illustrate some of them with the structure `trees` displayed in Sec. 2.8.16.

9.1.1 **Creating structures – structure(), strconcat() and split()** The basic command for creating a structure is `structure()`. Its general usage is `structure(c1,c2,...)`. The arguments, `c1`, `c2`, ..., are variables of any type, including `GRAPH` or other structures. The structure `trees` used in Sec. 2.8.16 was created by

```
Cmd> trees <- structure(info:"Made up data on 6 trees",\
  varnames:vector("Species","DBH"),\
  data:matrix(vector(1,1,1,2,2,2, 5.6,4.5,8.9,7.3,9.9,11.3),6))

Cmd> trees
component: info
(1) "Made up data on 6 trees"
component: varnames
(1) "Species"
(2) "DBH"
component: data
(1,1)           1         5.6
(2,1)           1         4.5
(3,1)           1         8.9
(4,1)           2         7.3
(5,1)           2         9.9
(6,1)           2        11.3
```

When, as here, an argument to `structure()` is a keyword phrase where the keyword is not `compnames`, `labels` or `warning`, the keyword specifies the name of a component. The name of a component that is not specified by a keyword phrase is the same as the name of the variable, with a starting "@" removed if it is a temporary variable (Sec. 2.4). When such an argument doesn't have a name because it is a computed quantity such as `sqrt(x)`, the component is given a descriptive name like `VECTOR` or `STRING`.

```
Cmd> @x <- 3;structure(run(5),sqrt_pi:sqrt(PI), @x)
component: VECTOR      Name given to run(5)
(1)           1            2           3           4           5
component: sqrt_pi     From keyword
(1)       1.7725
component: x           "@" stripped off @x
(1)           3
```

You can also name the components using a final argument `compnames:Names`, where `Names` is a `CHARACTER` vector or scalar. This is the only way to name a component `labels` or `compnames` with any name longer than 10 characters. When `Names` is a quoted string or `CHARACTER` scalar (single name), the component names all start with the `name` with "1", "2", ... appended; otherwise, `length(Names)` must match the number of components.

```
Cmd> hills <- structure(vector(2,3),vector(7,4),compnames:"hill")
```

```
Cmd> hills
component: hill1
(1)            2            3
component: hill2
(1)            7            4
```

In this example `hill` was used as a "root" from which `hill1` and `hill2` were constructed.

No element of `Names` may contain the character `$`.

In addition, you can *label* the components, possibly using longer names, using keyword `labels`.

```
Cmd> hills2 <- structure(vector(2,3),vector(7,4),\
    compnames:"hill",labels:"Minnesota Hill "); hills2
Minnesota Hill 1
(1)            2            3
Minnesota Hill 2
(1)            7            4

Cmd> getlabels(hills2)
(1) "Minnesota Hill 1"
(2) "Minnesota Hill 2"

Cmd> compnames(hills2) # component names as before (see Sec. 9.1.2)
(1) "hill1"
(2) "hill2"
```

See Sec. 8.4.1 for details on labels.

`strconcat()` provides another way to create a structure. It is used the same way as `structure()`. When none of its arguments is a structure, `strconcat()` behaves identically to `structure()`. However, when any argument is itself a structure, each of its top level components becomes a component of the result rather the argument being one component. Thus `strconcat()` allows you to combine two or more structures into one, without increasing the "depth."

```
Cmd> hills <- strconcat(hills,hill3:vector(9,5,11),hill3:12)

Cmd> hills
component: hill1
(1)            2            3
component: hill2
(1)            7            4
component: hill3
(1)            9            5           11
component: hill3
(1)           12
```

Note that, although it may be confusing, it is not illegal for component names to be duplicated (both components 3 and 4 have name `hill3`). However, `hills$hill3` would extract only component 3, the first component with name `hill3`. The only way to extract the second with name `hill3` (component 4) is to use a subscript:

```
Cmd> hills[4] # component 4, second hill3
(1)           12
```

If any arguments to `structure()` or `strconcat()` are functions (say `cos`), undefined variables, or simply missing, a warning message is printed and the value of the corresponding component of the output is `NULL`. You can suppress any warning message by including `silent:T` as an argument.

```
Cmd> structure(sin,cos)
WARNING: function name used as argument to structure()
component: sin
(NULL)
component: cos
(NULL)

Cmd> structure(sin,cos,silent:T)
component: sin
(NULL)
component: cos
(NULL)
```

Any of the keywords `compnames`, `labels` or `warnings` must follow all arguments that make up the components of the output.

`split()` is another tool for creating structures. Suppose `x` is a `REAL` or `LOGICAL` vector and the "splitting variable" `a` is a vector with the same length as `x`, with all the `a[i]` positive integers. Then `str <- split(x,a)` creates a structure with `max(a)` components, `comp1`, `comp2`, ..., with component *j* consisting of all `x[i]` for which `a[i]` has value *j*. In particular, if the splitting variable is a factor (see Sec. 3.3), component *j* of `str` consists of all `x[i]` at level *j* of that factor. If there are no values in `a` equal to *j*, component *j* is `NULL`.

```
Cmd> treatment <- factor(vector(1,1,1,2,2,2,2,3,3))

Cmd> z <- vector(7.7,10.7,10.2, 11.5,6.6,10.9,8.7, 8.7,9.2)

Cmd> split(z,treatment)
component: treatment1
(1)          7.7         10.7         10.2
component: treatment2
(1)         11.5          6.6         10.9          8.7
component: treatment3
(1)          8.7          9.2
```

It is also acceptable for the splitting argument to be a `LOGICAL` vector, in which case False and True correspond to factor levels 1 and 2, respectively.

```
Cmd> split(run(10),vector(F,T,T,F,F,T,T,T,F,T))
component: comp1       Corresponding to F
(1)            1            4            5            9
component: comp2       Corresponding to T
(1)            2            3            6            7            8
(6)           10
```

Any element of `x` corresponding to a `MISSING` value of the splitting variable is omitted from the results. It is an error for all the elements of the splitting variable to be `MISSING`.

If the splitting variable is not an expression but has a name, say `groups` or `@groups`, the components will be named `groups1`, `groups2`, etc. Similarly if the splitting variable is specified in a keyword phrase such as `dose:rep(run(4),5)`, components will be named `dose1`, `dose2`, … .

You can also use `split()` to split a matrix into its rows or columns, with each row or column becoming a component of a structure. When `x` is a matrix both `split(x)` and `split(x,bycols:T)` return a structure with one component for each *column* of x; `split(x,byrows:T)` returns a structure with one row vector component for each *row* of x.

```
Cmd> split(matrix(run(6),3)) # or split(matrix(run(6),3),bycols:T)
component: col1
(1)           1           2           3
component: col2
(1)           4           5           6
Cmd> split(matrix(run(6),3),byrows:T)
component: row1
(1,1)           1           4
component: row2
(1,1)           2           5
component: row3
(1,1)           3           6
```

You can use keyword `compnames` to name the components created by `split()`, just as with `structure()`.

An important usage for `split()` is to create an argument to a function that expects or accepts structures. Here we use `describe()` with an argument created by `split()` to compute the mean and the variance of `z` for each level of `a`.

```
Cmd> describe(split(z,treatment,compnames:vector("K","P","N")),\
    mean:T,var:T)
component: mean
  component: K
(1)       9.5333
  component: P
(1)        9.425
  component: N
(1)         8.95
component: var
  component: K
(1)       2.5833
  component: P
(1)       4.9958
  component: N
(1)        0.125
```

An particularly important use for `split()` is as an argument to `boxplot()` (see Sec. 2.12.2). `boxplot(split(x,a))` produces parallel box plots of `x` split by the levels of `a` and, when `x` is a matrix, `boxplot(split(x))` produces parallel box plots of the data in the columns of `x`. See Sec. 10.2 for an example of this usage.

9.1.2 **Getting information about a structure – ncomps() and compnames()** You may sometimes forget how many components there are in a structure or forget their names. When `str` is a structure, `ncomps(str)` returns the number of its components, and function `compnames(str)` returns a CHARACTER vector containing the names of those components. These functions are also very helpful in writing macros that manipulate structures (see Sec. 9.3).

```
Cmd> ncomps(trees)
(1)            3

Cmd> compnames(trees)
(1) "info"
(2) "names"
(3) "data"
```

9.1.3 **Changing a structure – changestr()** You can use `changestr()` to replace a component in a structure or add components to or delete components from a structure. `changestr()` does not actually change a structure directly but returns the modified structure as a value which can be assigned. Its usage is probably best illustrated by examples.

Modify a named component:

```
Cmd> changestr(trees,info:"New value for component info")
component: info
(1) "New value for component info" Component replaced
component: varnames
(1) "Species"
(2) "DBH"
component: data
(1,1)            1          5.6
(2,1)            1          4.5
(3,1)            1          8.9
(4,1)            2          7.3
(5,1)            2          9.9
(6,1)            2         11.3
```

If the structure has no component matching the given name, a new component with that name is added at the end of the structure.

```
Cmd> changestr(trees, date:"March 5, 1977")
component: info
(1) "Made up data on 6 trees"
component: varnames
(1) "Species"
(2) "DBH"
component: data
(1,1)            1          5.6
(2,1)            1          4.5
(3,1)            1          8.9
(4,1)            2          7.3
(5,1)            2          9.9
(6,1)            2         11.3
component: date
(1) "March 5, 1977"
```

The following do the same as these examples:

```
Cmd> changestr(trees,"info","New value for component info")
```

and

```
Cmd> changestr(trees,"date","March 5, 1977")
```

Modify a component referred to by number:

```
Cmd> changestr(trees,3,newdata:vector(7.1,5.1,3.7,2.8))
component: info
(1) "Made up data on 6 trees"
component: names
(1) "Species"
(2) "DBH"
component: newdata     Component 3 replaced and renamed
(1)          7.1          5.1          3.7          2.8
```

If the component number specified had been 4 instead of 3, then a new component would have been added.

Delete a component by specifying a negative number.

```
Cmd> changestr(trees,-3) # delete component 3
component: info
(1) "Made up data on 6 trees"
component: names
(1) "Species"
(2) "DBH"
```

Deleting a component is more easily done using a negative subscript, as in `trees[-3]`. It is illegal to delete the only component of a structure.

9.2 **Compound commands, conditional commands, and looping commands** You can group together several individual commands so that, for certain purposes, they are viewed as a single command known as a *compound command*. Syntax elements `if`, `else` and `elseif` permit conditional execution of compound commands, depending on the value of one or more `LOGICAL` variables or expressions. And syntax elements `while` and `for` allow "looping", that is, repetitive execution of compound commands.

9.2.1 **Compound commands** A compound command is a sequence of one or more ordinary commands or expressions surrounded by braces, that is, preceded by "`{`" and followed by "`}`". The individual commands making it up may be on separate lines or separated by semicolons. The value of a compound command is the value of the last individual command or expression in the sequence. Here is a compound command consisting of three individual parts.

```
Cmd> {@tmp <- 3*log(640320)/sqrt(163)
    @tmp + 2
    @tmp - PI}
(1)            0
```

Although the `@tmp + 2` is not an assignment, its value is *not* printed because it is part

of a compound command. In fact, it is there only to illustrate this point. The reason the value (0) of `@tmp - PI` *is* printed is because it is the last command before "`}`" making its value the value of the entire compound command. This value is printed because it is not assigned to a variable. In fact it would be printed even if the last command had been an assignment, say `diff <- @tmp - PI`, because the value of an assignment is the value assigned and this would become the value of the compound command. The value is 0 because $3 \times \log(640320)/\sqrt{163}$ is exactly π within rounding error.

Once you start a compound command by typing "`{`", MacAnova will continue to accept input until a closing "`}`" has been typed, even when the compound command extends over several lines. Under windowed versions (Macintosh, Windows, Motif) no prompt is given before lines other than the first.

```
Cmd> diff <- { # example on Windowed version
@tmp <- 3*log(640320)/sqrt(163)
@tmp; @tmp - PI
}; diff
(1)            0
```

The compound command is split among several lines and its value, namely the value of `@tmp - PI`, is assigned to variable `diff` which is printed outside the compound command.

In non-Windowed versions (Unix, DOS), the special prompt `More>` is printed to remind you that more input is needed as in the following.

```
Cmd> diff <- { # example on non-Windowed version
More> @tmp <- 3*log(640320)/sqrt(163)
More> @tmp; @tmp - PI
More> }; diff
(1)            0
```

You can *nest* compound commands. That is, a compound command may itself be made up of one or more compound commands, possibly together with non-compound commands. Here is a trivial example.

```
Cmd> {{x <- 3+4; y <- 7};{x <- 3*x; x+y}}# contains 2 compnd cmds.
(1)           28
```

The value of the entire compound command is the value of the second compound command, that, is the value of `x+y`.

You can force a compound command to have a `NULL` value by putting the explicitly null statement "`;;`" at its end.

```
Cmd> {{x <- 3+4; y<-7};{x <- 3*x};;} # value is NULL
```

9.2.2 **Conditional commands – if, elseif and else** Syntax element `if` allows execution of a compound command conditional on whether a `LOGICAL` variable or expression is True. The simplest form of a *conditional command* is

```
if(Logical) CompoundCommand
```

where `Logical` is a `LOGICAL` scalar variable or expression and `CompoundCommand` is a

compound command starting with "{" and ending with "}". The opening "{" *must be on the same line with* if. MacAnova first evaluates Logical; if its value is True then CompoundCommand is executed and the value of CompoundCommand becomes of the value of the conditional command; otherwise CompoundCommand is skipped and the conditional command has a NULL value.

```
Cmd> x <- 3; y <- 4

Cmd> if(x <= y){print("x <= y")}#Logical expression is True
x <= y

Cmd> if(x > y){print("x > y")}#Logical expression is False; no print

Cmd> b1 <- if(x < y){ #compound command on two lines
       2*x + 10} # value assigned is non-NULL

Cmd> b2 <- if(x > y){2*x + 10} # value assigned is NULL

Cmd> print(b1, b2)
b1:
(1)           16
b2:
(NULL)
```

The enclosing braces are required even when the compound command consists of only one command or expression. See below.

Use of syntax element else with if allows you to direct MacAnova to do one thing if the LOGICAL expression is True and something else if it is False. The general usage is the conditional command

```
if(Logical) CompoundCommand1 else CompoundCommand2
```

where *both* CompoundCommand1 and CompoundCommand2 start and end with "{" and "}". The opening "{" of CompoundCommand1 must be on the same line with if and both the closing "}" of CompoundCommand1 and the opening "{" of CompoundCommand2 must be on the same line with else.

In an if … else … conditional command, MacAnova first evaluates Logical. If its value is True then CompoundCommand1 is executed and CompoundCommand2 is skipped; if its value is False than CompoundCommand1 is skipped and CompoundCommand2 is executed. The value of an if … else … conditional command is the value of the compound command actually executed. For example, the value of if(T){1}else{2} is 1, while the value of if(F){1}else{2} is 2.

```
Cmd> x <- 3; y <- 4

Cmd> if (x < y){print("x < y")}else{print("x >= y")}
x < y          Printed by 1st compound command

Cmd> signdiff <- if (x < y){-1}else{1};signdiff # assigned value
(1)           -1

Cmd> x <- 4; y <- 4

Cmd> if (x < y){print("x < y")}else{print("x >= y")}
x >= y         Printed by 2nd compound command
```

```
Cmd> signdiff <- if (x < y){-1}else{1};signdiff # assigned value
(1)            1
```

You can specify more than two choices using `elseif`.

```
if(Logical1) CompoundCommand1 elseif(Logical2) CompoundCommand2\
  else CompoundCommand3
```

Now `CompoundCommand1` is executed if `Logical1` is True, `CompoundCommand2` is executed when `Logical1` is False and `Logical2` is True, and `CompoundCommand3` is executed if both `Logical1` and `Logical2` are False. The value of an `if … elseif … else …` conditional command is the value of the compound command actually executed. For example, the value of `if(T){1}elseif(T){2}else{3}` is 1, the value of `if(F){1}elseif(T){2}else{3}` is 2 and the value of `if(F){1}elseif(F){2}else{3}` is 3. You can have additional `elseif` pieces in a conditional command before the `else` piece, and the `else` piece can be omitted.

```
Cmd> x <- 1; y <- 7

Cmd> if(x < y){tmp <- y - x;;} else {tmp <- x - y;;}; tmp
(1)            6
```

Here the `LOGICAL` expression is True and the first compound command is executed. The `if … else` line is actually computing and printing the value of `abs(x-y)`. In this case, the value of the conditional command is the value of `{tmp <- y - x;;}` because that is the compound command actually executed. Because of the extra "`;;`", the value is `NULL` and hence is not printed. Because a conditional command has a value, an alternative line doing the same thing is as follows:

```
Cmd> tmp <- if(x < y){y-x}else{x-y} ; tmp # Assignment to tmp
(1)            6
```

Because the `LOGICAL` expression `x < y` is True, `tmp` is assigned the value, `y - x`, of the first compound command; in the contrary case it would be assigned the value, `x–y`, of the second compound expression.

```
Cmd> x <- 3; if(x > 0) {1} elseif (x < 0) {-1} else {0}
(1)            1

Cmd> x <- -3;if(x > 0) {1} elseif (x < 0) {-1} else {0}
(1)           -1

Cmd> x <- 0; if(x > 0) {1} elseif (x < 0) {-1} else {0}
(1)            0
```

The conditional command is the same in all three lines, in each of which a different compound command is executed.

9.2.3 **Looping – for and while** Sometimes you may need to repeat one or more commands several times. You could just type them in again and again, but it is sometimes easier to "program" a *loop* that keeps going back and executing the same commands several times. In MacAnova there are two kinds of loops, `while` loops and `for` loops.

A `while` loop has the form

```
while(Logical) CompoundCommand
```

where, as usual, CompoundCommand starts with "{" and ends with "}" and may extend over several lines. The opening "{" *must be on the same line with* `while`. When `Logical`, say `n > 0`, has value True, CompoundCommand is executed. Then `Logical` is again evaluated and if it is still true CompoundCommand is again executed. This continues as long as the value of `Logical` is True. If and when `Logical` is false, CompoundCommand is skipped. It is essential that something happens in CompoundCommand that eventually changes the value of `Logical`, say `n <- n-1`; if not, the loop cannot terminate properly, that is, is an *infinite loop*. To be on the safe side, MacAnova terminates `while` loops after 1000 repetitions, whether or not `Logical` has become False.

```
Cmd> s <- 0; n <- 10; while(n > 0) { s <- s + n; n <- n-1;;}; s
(1)           55    Sum 10+9+8+7+6+5+4+3+2+1

Cmd> s <- 0; n <- 10; while(n > 0) {
    s <-s  + n;; # Note that n is not decremented
}
ERROR: more than 1000 repetitions of while loop
```

Nothing in the compound command in the second loop changes `n` so the loop would go on forever if MacAnova did not lose patience. Note also the null command ";;" before the closing "}" to prevent output on each trip through the loop.

The default maximum number of repetitions can be changed by option `maxwhile` to any value 10 or above (see Sec 8.1.3).

```
Cmd> setoptions(maxwhile:50) # only 50 trips through loop allowed

Cmd> i <- 0; while(i >= 0){i <- i+1;;} # i incremented on every trip
ERROR: more than 50 repetitions of while loop

Cmd> i
(1)           50
```

A `for` loop has the general form

```
for(index, Range) CompoundCommand
```

where `index` is a legal variable name and `Range` is a `REAL` vector. First `index` is set to `Range[1]` and CompoundCommand is executed; then `index` is set to `Range[2]` and CompoundCommand is again executed; and so on for each element in `Range`. The most common form for `Range` is probably `run(n)`, where `n` is an integer.

```
Cmd> s <- 0; for(i,run(5)){s <- s + i;;}; s # s <- 1+2+3+4+5
(1)           15
```

Since `length(run(5))` = 5, the compound command is executed 5 times and `i` successively takes values 1, 2, 3, 4, and 5. This line is effectively equivalent to

```
s <- 0;s <- s+1;s <- s+2;s <- s+3;s <- s+4;s <- s+5; s
```

If `Range` is `NULL`, CompoundCommand is skipped entirely.

Another form for Range is probably best illustrated by examples:

```
Cmd> s <- 0; for(i,3,10){s <- s + i;;}; s #3+4+5+6+7+8+9+10
(1)           52

Cmd> s <- 0; for(i,1,3,1/3){s <- s + i;;};s #1+4/3+5/3+2+7/3+8/3+3
(1)           14
```

for(index,i1,i2) is equivalent to for(index,run(i1,i2)) and for(index,i1,i2,inc) is equivalent to for(index,run(i1,i2,inc)) (see Sec. 2.14).

Unlike while loops, there is no fixed limit for the possible number of repetitions of a for loop – it depends on the length of Range.

9.2.4 **Escaping from loops – break and breakall** In some circumstances you may want to terminate a while loop before the logical variable becomes False or leave a for loop before all the elements in the range vector have been used up. This is possible using syntax element break. In the following we attempt to evaluate the geometric series $1 + x + x^2 + x^3 + \ldots = 1/(1-x)$ when $|x| < 1$, terminating the loop when there is no point in computing further terms.

```
Cmd> x <- .57 # small enough to converge rapidly

Cmd> s <- 1; for(i,run(30)){
    term <- x^i; s <- s + term
    ratio <- abs(term/s)
    if(ratio < .000001){break}
}

Cmd> vector(i, ratio, s, 1/(1-x)) #converged in 24 trips around loop
(1)           24   5.9493e-07       2.3256       2.3256

Cmd> x <- .8 # larger value of x; slower convergence

Cmd> s <- 1; for(i,run(30)){
    term <- x^i; s <- s + term
    ratio <- abs(term/s)
    if(ratio < .000001){break}
}

Cmd> vector(i, ratio, s, 1/(1-x))
(1)           30   0.00024783        4.995           5
```

Because length(run(30)) = 30, the compound statements will be executed at most 30 times. When the ratio ratio of a term to the current sum s becomes small enough (< .000001), break exits the loop, skipping over everything up to and including the closing "}". In the first case this happened when i was 24 and so the loop was executed only 24 times. In the second case, break was not executed and the loop ran the full 30 times; because the convergence criterion was not satisfied, the sum 4.995 is about 0.1% below the correct value $1/(1-.8) = 5$.

It is possible to have *nested* loops, that is have one loop inside another. You can exit from several for or while loops at once by adding a literal integer after break. For example, break 2 and break 3 exit from two and three looping levels, respectively, while break 1 means the same thing as break. The following example illustrates the use of nested for loops to determine if any elements of a matrix are 0.

```
Cmd> a <- matrix(vector(1,3,4,2,5,0,6,7),2); a # has 1 zero
(1,1)           1           4           5           6
(2,1)           3           2           0           7

Cmd> foundzero <- F

Cmd> for(i,run(nrows(a))){
        for(j,run(ncols(a))){
            if(a[i,j] == 0){foundzero <- T; break 2}
        }
    }

Cmd> if(foundzero){
        print(paste("a[",i,",",j,"] == 0",sep:""))
    } else {
        print("No element of a is 0")
    }
a[2,3] == 0        Output from first print()
```

Here we have a `for` loop over variable `j` nested within a `for` loop over `i`. When a zero element of `a` is found, `break 2` terminates *both* loops . If you replaced `break 2` by `break` or `break 1`, it would terminate the inner loop only. Even if `n` were a variable with value 2, `break n` would not be legal. An alternative to the use of `break 2`, is `breakall` which exits from *all* loops, but its use can lead to problems that are difficult to diagnose. For the use of `paste()`, see Sec. 8.3.1.

9.2.5 **Skipping to the end of a loop – next** It's sometimes helpful to be able to skip to the end of a loop without terminating it. You can do this using syntax element `next`. This has almost the same effect as using `break` except that after skipping the rest of the loop, execution resumes just before the "`}`" that terminates the loop instead of after it. Here is an example based on the first example in Sec. 9.2.5 in which the 100 terms of the series are summed as well as the "tail" of the series after a convergence criterion is satisfied.

```
Cmd> x <- .57; s1 <- 0; s <- 1; for(i,run(100)){
        term <- x^i;
        ratio <- abs(term/s)
        s <-+ term
        if(ratio >= .000001){next}
        s1 <-+ term ;;# executed only when ratio < .000001;;
    }

Cmd> vector(s, s1) # converged value and remainder
(1)       2.3256   3.2176e-06

Cmd> x <- .80; s1 <- 0; s <- 1; for(i,run(100)){
        term <- x^i;
        ratio <- abs(term/s)
        s <-+ term
        if(ratio >= .000001){next}
        s1 <-+ term ;;# executed only when ratio < .000001;;
    }

Cmd> vector(s, s1) # converged value and remainder
(1)            5   2.3383e-05
```

You can to skip to the end of loop from inside a loop that is nested within it by using a literal integer after next. For example, next 1 means the same thing as next; next 2 leaves the current loop and skips to the end of the loop enclosing it; next 3 leaves the current loop and the one enclosing it and skips to the end of the next higher level loop; and so on. The integer must be a literal 1, 2, ... and not a variable with an integer value. Thus n <- 2; next n is always illegal. Here is a simple example to count the number of matrix rows which contain 0.

```
Cmd> a <- matrix(vector(1,3,4,2, 5,0,6,7, 9,10,0,8),3) #3 by 4

Cmd> rowswithzero <- 0 # initialize count

Cmd> for(i,run(nrows(a))){ # loop over rows
       for(j,run(ncols(a))){ # loop over columns within row
          if(a[i,j] == 0){rowswithzero <-+ 1; next 2}
       }
   } # next 2 skipped to just before this '}'

Cmd> rowswithzero # number of rows with zeros
(1)            2
```

9.3 **Macros** Although you use macros almost identically to functions and commands, a *macro* is actually a special type of variable somewhat similar to a CHARACTER scalar. Its value, the *text* of the macro, is made up of one or more MacAnova commands all grouped together. When MacAnova encounters a macro name followed by a list of arguments separated by commas enclosed in (...), it executes the commands making up the macro one after another, possibly printing some results or returning a value, just like a function.

It is actually is a bit more complicated than that. Before executing the commands, the macro is *expanded*. First, at appropriate places, MacAnova inserts the macro's arguments in the text of the macro (Sec. 9.3.2). Then it scans the text for special symbols starting or ending with "$" which get special treatment (Sec. 9.3.4). Finally it puts "{" and "}" at the start and at the end, turning the sequence of commands into a compound command. This compound command is then executed exactly as if you had typed it. Macros may be expanded in-line (the default) or out-of-line. See Sec. 9.3.5.

The value of the executed macro is the value of the last command in it. This is the key to understanding how a macro returns a value to be printed or assigned to a variable. A macro that is not intended to return a value should return a NULL value. If the last command does not itself return a NULL value as do the GLM commands and output commands such as print(), it should be followed by ";;" (see Sec. 9.2.1) or simply the constant NULL (see Sec.2.5).

It may sometimes be important to know that each instance of an in-line macro in a command line is expanded only once, even if it is in a loop and is executed several times. The second and any subsequent times it is executed, it has already been expanded and is executed as is.

There are several pre-defined macros such as readcols and getdata that are described elsewhere (Sec. 2.11.2 and 2.11.4). For most purposes you can use these just like functions and even ignore the fact that they are macros. However, since having the

right macro can be a great labor saving device (you don't need to type all the commands in the macro separately, just the macro name and argument list), many users will at some point want to write their own macros. Using macros you can almost indefinitely extend the statistical or mathematical analyses MacAnova can do. This section summarizes what you need to know to write macros.

9.3.1 **Creating macros** You use function `macro()` to create a macro. It has a single quoted string or `CHARACTER` variable as its argument.

```
Cmd> mymac <- macro("$1 * $1 + $2") # create macro mymac

Cmd> list(mymac) # it's a variable of type MACRO
mymac           MACRO  (in-line)
```

In many respects, a macro is like a `CHARACTER` variable. In particular, if you type its name, the macro will be printed.

```
Cmd> mymac # print it
(1) "$1 * $1 + $2"
```

See Sec. 9.3.3 and Sec. 9.3.5 for using keyword phrases `dollars:T` and `inline:F`, respectively, as additional arguments to `macro()`. See Sec. 9.3.5 for a discussion of the difference between in-line and out-of-line macros.

Macros can also be read from external files by function `macroread()` (Sec. 7.5.1). Pre-defined macro `getmacros` (Sec. 7.5.3) uses `macroread()` to read macros from the files whose names are specified in `CHARACTER` vector `MACROFILES`.

Because `macroread()` can also read from the special `CHARACTER` variable `CLIPBOARD` (Sec. 7.3), in versions with windows (Macintosh, Windows, Motif), you can edit a macro in another program or even a MacAnova command/output window, and then copy it to the clipboard, from which it is read by `macroread()`.

See Sec. 8.8.3 for a way to create a macro from recently executed commands.

9.3.2 **Argument substitution** The most important special symbols starting with "$" are the "place holders" `$1`, `$2`, `$3`, ... MacAnova literally replaces `$1` by the characters making up argument 1, replaces `$2` by the characters making up argument 2, and so on. This is the way MacAnova knows where the arguments are to be inserted. Macro `mymac` in Sec. 9.3.1 is apparently intended to compute the product of the first argument with itself and add the product to the second argument. Since `$1 * $1 + $2` is the last (and only) statement in the macro, the value of `mymac` is the value of `$1 * $1 + $2` after substituting the arguments for `$1` and `$2`.

```
Cmd> mymac(4,5) # expands as {4 * 4 + 5}
(1)           21       Value gets printed

Cmd> mymac(4,run(3)) # expands as {4 * 4 + run(3)}
(1)           17           18           19
```

This simple macro has been poorly written, however, and sometimes doesn't work the way intended.

```
Cmd> mymac(3+1,run(3)) # expands to {3+1 * 3+1 + run(3)}
(1)            8            9           10
```

This gives a different answer from `mymac(4,run(3))`. What went wrong? The problem is that the arguments were substituted *literally*, to produce the expression `3+1 * 3+1 + run(3)`. But, because multiplication has a higher precedence than addition (see Sec. 2.8.3), MacAnova interpreted this as `3+(1*3)+1+run(3)`, not as `(3+1)*(3+1) + run(3)`. Fortunately, you can easily correct this problem by surrounding each argument place holder with parentheses as in this revised version of `mymac`.

```
Cmd> mymac <- macro("($1) * ($1) + ($2)") # $1 & $2 in parentheses

Cmd> mymac(3+1,run(3)) # expands as {(3+1) * (3+1) + (run(3))}
(1)           17           18           19
```

Substitutions are made for `$1`, `$2`, ... even when they are enclosed in quotation marks.

```
Cmd> printmsgs <- macro("print(\"$1\")")

Cmd> printmsgs # Macro to print its argument as string
(1) "print(\"$1\")"

Cmd> printmsgs(MacAnova's great)
MacAnova's great
```

This works because `printmsgs(MacAnova's great)` expands to `{print("MacAnova's great")}`.

There is one exception to the exact substitution of arguments. When (a) an argument itself contains '"' or '\', and (b) it is to be substituted inside quotes (`"..."`) in the macro itself, then, '"' and '\' are replaced by '\"' and '\\', respectively, when the argument is expanded.

```
Cmd> printmsgs("MacAnova's great")
"MacAnova's great"
```

The macro expanded to `{print("\"MacAnova's great\"")}`.

If you don't supply enough arguments when using a macro, it may cause an error. If the missing argument is inside a quoted string, it "expands" to nothing. Otherwise, if the missing argument is ever referenced, an error message is printed.

```
Cmd> printarg1 <- macro("print(\"Argument 1 is '$1'\")")

Cmd> printarg1()
Argument 1 is ''

Cmd> print2 <- macro("print($1,$2)") # requires two arguments

Cmd> print2(PI) # one argument only
ERROR: Argument 2 to macro print2 missing
```

If you include a leading 0 in the place holder (`$01`, `$02`, ...), then a missing argument outside of quotes is expanded to `NULL`.

```
Cmd> print2a <- macro("print($01,$02)") # expects two arguments
```

```
Cmd> print2a(PI) # one argument;expands to {print(PI,NULL)}
NUMBER:
(1)       3.1416
NULL:       Argument 2 is taken to be NULL
(NULL)
```

9.3.3 **The use of temporary variables and $$** Although it now works, our example macro mymac still has a possible problem. Consider the following

```
Cmd> mymac(sum(log(run(2000)))/2000,exp(run(3)))
(1)       46.321       50.992       63.689
```

The answer is correct. However, the macro expands to the compound command

```
{(sum(log(run(2000)))/2000)*(sum(log(run(2000)))/2000)+(exp(run(3)))}
```

In the process of computing this, `sum(log(run(2000)))/2000` is evaluated *twice*. Although this is not a big deal, it results in loss of efficiency, which on a slow computer would be noticable. Even on a very fast computer, similar inefficiencies can be important. Here is one solution to the problem:

```
Cmd> mymac <- macro("@x <- $1
    @x * @x + ($2)")

Cmd> mymac
(1) "@x <- $1
    @x * @x + ($2)"

Cmd> mymac(sum(log(run(2000)))/2000,exp(run(3)))
(1)       46.321       50.992       63.689
```

First, note that mymac is now a two line macro, with the result being the value computed on the last (second) line. Secondly, observe that mymac now sets a temporary variable @x to the value of the first argument. Since this is the only place that $1 is referenced directly, `sum(log(run(2000)))/2000` is computed only once. Because its name starts with "@", @x is a temporary variable that is deleted no later than the next prompt. As a general rule, it is usually a good idea to copy any argument referred to more than once to a temporary variable. An exception might be when you know the argument may be a very large matrix, in which case making a copy might use up too much computer memory.

There is still at least one more possible problem in this macro. Consider the following macro mymac2 which invokes mymac, switching the order of its arguments.

```
Cmd> mymac2 <- macro("@x <- $1; @y <- $2;mymac(@y, @x)")

Cmd> mymac2
(1) "@x <- $1; @y <- $2;mymac(@y, @x)"

Cmd> mymac(4,run(3))
(1)          17           18           19

Cmd> mymac2(run(3),4)
(1)          20
```

Since mymac2 simply invokes mymac with its arguments in reverse order, you would expect `mymac2(run(3),4)` to be the same as `mymac(4,run(3))`, but it clearly is not.

What is going wrong? The problem is that both macros use the *same* temporary variable @x. Here is what the full expansion of mymac2(run(3),4) looks like, including the expansion of mymac.

```
{@x <- run(3);@y <- 4;{@x <- @y
@x * @x + (@x)}}
```

The italicized part is the expansion of mymac and the bold face variables are the substitutions in mymac. It is not doing what we want because mymac is changing the value of @x set by mymac2 before it gets around to using it. You might say that whoever wrote mymac2 should have known better than to use a temporary name that is also used in mymac. However, you shouldn't have to know the inner details of a macro in order to use it safely.

There is a way to avoid such a conflict of temporary names. Here are new versions of mymac and mymac2. In them we use special names @x$$ and @y$$ instead of simply @x and @y.

```
Cmd> mymac<-macro("@x$$ <- $1
@x$$ * @x$$ + ($2)") # note the trailing "$$"'s

Cmd> mymac2 <- macro("@x$$ <- $1; @y$$ <- $2
mymac(@y$$, @x$$)") # trailing "$$"'s used again

Cmd> mymac2(run(3),4) # now it's correct
(1)          17          18          19
```

To see what is happening, here is the complete expansion of the new version of *mymac2(run(3),4)*:

```
{@x50 <- run(3);@y50 <- 4
{@x51 <- @y50
@x51 * @x51 + (@x50)}}
```

where italics and boldface mean the same as before. The trailing "$$" has been expanded as "50" in mymac2, but as "51" in mymac. Since @x50 and @x51 are different names, there is no conflict. Generally a trailing "$$" is expanded to a unique number between 50 and 99 in each in-line macro and between 00 and 49 in each out-of-line macro.

You don't explicitly have to add "$$" to temporary variable names if you use keyword phrase dollars:T as an argument to macro() when creating a macro.

```
Cmd> mymac <- macro("@x <- $1
@x * @x + ($2)",dollars:T)

Cmd> mymac # "$$"'s were automatically added
(1) "@x$$ <- $1
@x$$ * @x$$ + ($2)"
```

In summary, there are two rules to observe in writing safe macros:

(a) Enclose $1, $2, ... in parentheses except in quoted strings

(b) Use names starting with "@" and ending with "$$" for temporary variables.

See Sec. 9.3.6 for a recommendations on deleting temporary variables in a macro.

9.3.4 **Other expanding macro symbols** The special macro symbol $0, $N, $V, $v, $K, $k, $A, and $S all are replaced as part of macro expansion.

Macro symbol $0 is replaced by the entire argument list, *including the commas* that separate the multiple arguments. This is particularly helpful for writing an "alias" for a function, that is a macro which does the same thing as the function but has a different name. For instance, a DOS user who is used to typing dir to get a list of files, might wish command list() was named dir(). No sooner said than done:

```
Cmd> dir <- macro("list($0)") # create 'alias' for command list()

Cmd> dir(real:T)  # this expands to {list(real:T)}
DELTAT          REAL   1
PI              REAL   1
```

Another use for this feature is illustrated by yet another version of macro mymac:

```
Cmd> mymac <- macro("@args$$ <- structure($0)
@args$$[1] * @args$$[1] + @args$$[2]")

Cmd> mymac(4,run(3))
(1)           17           18           19
```

The first line of this macro expands to @args50 <- structure(4,run(3)) which creates a structure @args50 with two components which are referred to by number in line 2 (see Sec. 2.8.16, 9.1.1).

Macro symbol $N is replaced by the number of arguments to the macro in the current invocation.

```
Cmd> testN <- macro("paste(\"The number of arguments is\",$N)")

Cmd> testN()
(1) "The number of arguments is 0"

Cmd> testN(1,"a",T,last:4)
(1) "The number of arguments is 4"
```

(For the use of paste() see Sec. 8.3.1)

Macro element $V is similar to $0 except that it is replaced by a comma-separated list of all arguments that are *not* keyword phrases. Macro element $v is replaced by the number of such arguments.

```
Cmd> testV <- macro("print(Dollarv:$v,DollarV:structure($V))")

Cmd> testV(1,"a",tau:4,T) # 3 non-keyword arguments
Dollarv:
(1)           3       Number of non-keyword arguments
DollarV:              Structure with all three arguments
component: NUMBER
(1)           1
component: STRING
(1) "a"
component: LOGICAL
(1) T
```

In parallel with $V and $v, macro elements $K is replaced by a comma-separated list of

the *keyword* arguments and $k by the number of such arguments.

```
Cmd> testK <- macro("print(Dollark:$k,DollarK:structure($K))")

Cmd> testK(1,"a",tau:4,T) # 1 keyword phrase argument
Dollark:
(1)           1     Number of keyword arguments
DollarK:
component: tau
(1)           4
```

Macro element $A is entirely equivalent to vector("$1","$2",...), which expands to a CHARACTER vector containing the text of each argument in quotation marks. Any cases of '"' or '\' in an argument are replaced by '\"' and '\\'.

```
Cmd> testA <- macro("@A <- $A # character version of arguments
@args <- structure($0) # ordinary version of arguments
for(@i,run($N)){
    print(paste(@A[@i],\"=\",@args[@i]))
}",dollars:T) # 5 line macro

Cmd> testA(3+4, PI, sqrt(20))
3+4 = 7
PI = 3.1416
sqrt(20) = 4.4721
```

Finally, macro element $S expands to the name of the macro.

```
Cmd> testS <- macro("print(\"This is macro $S\")")

Cmd> testS1 <- testS # copy testS to testS1

Cmd> testS()
This is macro testS

Cmd> testS1() # identical to testS except for its name
This is macro testS1
```

All these special "$" macro elements except $A are expanded whether or not they are inside a quoted string in the macro. $A is expanded only when it is *not* in a quoted string. Furthermore, none of $N, $V, $v, $K, $k, $A or $S is expanded when it immediately follows a legal MacAnova name and thus could refer to a structure component. For example, even in a macro, data$N is assumed to refer to a component named N of a structure named data.

9.3.5 **In-line and out-of-line macros** There are two expansion modes for macros – in-line and out-of-line. When a macro is expanded in-line, the macro "call" is actually *replaced* by the expanded text of the macro. For example, if testS is the macro defined in Sec. 9.3.4, the command line

```
print("Pre");testS();print("Post")
```

becomes

```
print("Pre");{print("This is macro testS")};print("Post")
```

once testS() is reached.

Because the original macro call is replaced, an in-line macro in a loop is expanded only once, the first time through the loop. This means that even if the macro is redefined during the loop, the change has no effect until the loop is ended. Consider the following example in which macro printi is defined 3 times to be successively "print(i:1)", "print(i:2)" and "print(i:3)". Because it is expanded only once, only the first form is operative.

```
Cmd> for (i, run(3)){
    printi <- macro(paste("print(i:",i,")",sep:""),inline:T)
    printi()
}
i:
(1)           1
i:
(1)           1
i:
(1)           1
```

Here we have used keyword phrase inline:T to ensure expansion is in-line. This is necessary only when *option* inline has value False (see Sec. 8.1.3). See Sec. 8.3.1 for the use of paste().

When an out-of-line macro is expanded, the original command line is not changed. Instead, the macro is expanded elsewhere, the expanded text is executed, and then execution resumes immediately after the macro call. These means that an out-of-line macro in a loop is expended every time through the loop. Here is the same example, except that printi is created as an out-of-line macro using keyword phrase inline:F.

```
Cmd> for (i, run(3)){ # note the use of inline:F
    printi <- macro(paste("print(i:",i,")",sep:""),inline:F)
    printi()
}
i:
(1)           1
i:
(1)           2
i:
(1)           3
```

Now printi is expanded out-of-line each time through the loop so the updated version is executed.

There is little reason to use out-of-line macros unless you want to do something like this example does – change a macro every time through a loop.

9.3.6 **Using delete(result,return:T) in a macro** Although temporary variables created in a macro are automatically deleted the next time the prompt is printed (Sec. 2.4), they survive until then, occupying space in memory. For this reason, it is good practice to "clean up" at the end of a macro, using delete() to remove temporary variables created in the macro (see Sec. 2.8.9). Of course, if one of the temporary variables, say @value, contains the result that the macro is returning, you won't want to include it with other variables as an argument to delete(). Instead, the last line of the macro should be

```
delete(@value, return:T)
```

Variable @value will be deleted, but its value will be returned as the value of delete(). Since this is the last expression in the macro, its value will be returned as the value of the macro. Here is the text of a simple macro using this feature.

```
@a$$ <- $1; @b$$ <- $2
@x$$ <- invbeta(run(999)/1000,@a$$,@b$$)
delete(@a$$, @b$$) # clean up
delete(@x$$, return:T) # @x$$ is deleted, its value returned
```

You can also use this feature to put a temporary variable in the argument list of a function while at the same time deleting it. Here is a variant on the previous macro.

```
@a$$ <- $1; @b$$ <- $2
@x$$ <- invbeta(run(999)/1000, delete(@a$$,return:T),\
    delete(@b$$,return:T))
delete(@x$$, return:T)
```

No more than one variable can be deleted when return:T is used with delete().

9.4 **Functions useful in macros** Although the commands in this section are most likely to be used in macros, they can be used at the prompt level, too. paste(), described in Sec. 8.3.1, 8.3.2 and 8.3.3 is also very useful in macros.

9.4.1 **Functions unique() and match()** unique() extracts the unique elements of a vector, that is every distinct element. When keyword phrase index:T is an argument, index() returns a vector of integers such that v[unique(v,index:T)] is equivalent to unique(v).

```
Cmd> v <- vector(3.1,2.5,2.5,4.3,3.7,6.8,6.8,3.1);unique(v)
(1)         3.1         2.5         4.3         3.7         6.8
```

Although 3.1 and 6.8 appear twice in v, they appear only once in unique(v).

```
Cmd> unique(v,index:T)
(1)           1           2           4           5           6
```

match() allows you to compare for equality each element of its first argument with each element of a vector second argument. You can use it to test whether a REAL or CHARACTER vector contains a specified value.

Suppose x is a scalar, v is a vector, and noMatch is a scalar and they all have the same type, either REAL or CHARACTER. Then the value of match(x,v,noMatch) is noMatch when no element of v is the same as x, and is otherwise k, where v[k] is the first element in v that is the same as x.

```
Cmd> match(6.8,v,-1)
(1)           6   6th element of w is 6.8

Cmd> match(7.0,v,-1)
(1)          -1   No element of w is 7.0
```

If x is a REAL or CHARACTER vector, matrix, or array, match(x,v,noMatch) returns a vector of the same type, shape and size as x. The i, j, ... element of the result is

match(x[i,j,...],v,noMatch). If any element of x is MISSING, the corresponding element of the result is also MISSING.

```
Cmd> match(vector(6.8,7.0),v,-1) # 6.8 matches, 7.0 does not
(1)           6          -1
```

```
Cmd> a <- factor(match(v,sort(unique(v))));a # create factor from v
(1)           2           1           1           4           3
(6)           5           5           2
```

Pre-defined macro makefactor (see Sec. 3.3) uses a command similar to this last example.

match(x,v), omitting noMatch, does the same with a default value of noMatch = length(v)+1, but prints a warning message if any element of x is not matched.

```
Cmd> match(vector(6.8,7.0),v)
WARNING: 1 values not matched coded as 9
(1)           6           9
```

When x and v are CHARACTER variables, and any of the elements of x contain the "wild card" characters "*" or "?", you can use match(x,v [,noMatch],exact:F) to determine which, if any, elements of v match the "patterns" specified by x. A "*" in x matches *any* sequence of 0 or more characters in an element of v and a "?" matches *any* single character in an element of v.

```
Cmd> v1 <- vector("abc","ade","gfh")
```

```
Cmd> match(vector("*c","*d*","g*","g*h", "a*b*c"), v1, exact:F)
(1)           1           2           3           3           1
```

```
Cmd> v2 <- vector("aqbde","bb123", "allbdef")
```

```
Cmd> match(vector("a*b???","a*b??"),v2,exact:F)
(1)           3           1
```

One use of match() is in selecting rows and / or columns corresponding to specified values of the coordinate labels of a labeled variable (see Sec. 8.4). Here is a short example:

```
Cmd> y <- matrix(10*run(2)+run(3)',2,labels:structure("R","C"))
```

```
Cmd> y # note the row and column labels
            C1          C2          C3
R1          11          12          13
R2          21          22          23
```

```
Cmd> y[match("R2",getlabels(y,1)),\
       match(vector("C1","C3"),getlabels(y,2))]
            C1          C3
R2          21          23
```

```
Cmd> # This is equivalent to y[2,vector(1,3)]
```

9.4.2 **Checking the characteristics of variables – isxxxx() functions** There are several MacAnova functions whose primary use is in checking to see that arguments to a macro are as expected. Functions isarray(), ischar(), isdefined(), isfactor(), isgraph(), islogic(), ismacro(), ismatrix(), isreal(), isscalar(),

`isstruc()`, and `isvector()` all return a `LOGICAL` vector whose length is the number of function arguments. The *i*-th element of the result is True if and only the the *i*-th argument satisfies the condition specified by the function name. In the following, `UnDef` is not the name of any existing variable or macro. Also note that "?" represents a `REAL MISSING` value.

```
Cmd> isreal(3,"MacAnova",run(3),?,3 < 4,structure(1,2))
(1) T       F       T       T       F       F

Cmd> islogic(3,"MacAnova",run(3),?,3 < 4,structure(1,2))
(1) F       F       F       F       T       F

Cmd> ischar(3,"MacAnova",run(3),?,3 < 4,structure(1,2))
(1) F       T       F       F       F       F

Cmd> isstruc(3,"MacAnova",run(3),?,3 < 4,structure(1,2))
(1) F       F       F       F       F       T

Cmd> ismacro(PI,boxcox)
(1) F       T

Cmd> a <- vector(1,1,2,2,3,3); b <- factor(a)

Cmd> isfactor(a,b)
(1) F       T

Cmd> isdefined(PI,NULL,UnDef)
(1) T       T       F

Cmd> isscalar(UnDef,3,run(3),matrix(run(4),2),array(run(8),rep(2,3)))
(1) F       T       F       F       F

Cmd> isvector(UnDef,3,run(3),matrix(run(4),2),array(run(8),rep(2,3)))
(1) F       T       T       F       F

Cmd> ismatrix(UnDef,3,run(3),matrix(run(4),2),array(run(8),rep(2,3)))
(1) F       T       T       T       F

Cmd> isarray(UnDef,3,run(3),matrix(run(4),2),array(run(8),rep(2,3)),\
    boxcox,structure(PI))
(1) F       T       T       T       T       F       F

Cmd> isnull(NULL,print("Value of print() is NULL"),1,T,?)
Value of print() is NULL
(1) T       T       F       F       F
```

`isscalar()`, `isvector()`, `ismatrix()` and `isarray()` allow simultaneous testing for shape and type using keywords `real`, `logic`, and `char`. Here are examples with `isscalar()`.

```
Cmd> realscalar <- PI;logicscalar <- T; charscalar <- "A"

Cmd> isscalar(realscalar,logicscalar,charscalar,real:T)
(1) T       F       F

Cmd> isscalar(realscalar,logicscalar,charscalar,logic:T)
(1) F       T       F

Cmd> isscalar(realscalar,logicscalar,charscalar,character:T)
(1) F       F       T
```

These can be extremely helpful in thoroughly checking macro arguments for appropriateness.

Function ismissing(), described in Sec. 2.7, does not fit this pattern. It has one argument and ismissing(x) returns a LOGICAL value the same shape as x with True in every position where an element of x is MISSING and False everywhere else.

```
Cmd> ismissing(matrix(vector(1,?, ?,4, 5,6),2))
(1,1) F       T       F
(2,1) T       F       F
```

When writing a macro, especially one that expects keyword phrases as arguments, using these isxxxx functions to check arguments can be complicated. The more specialized functions, keyvalue() and argvalue() may be easier to use. See Sec. 9.4.5 and 9.4.6.

9.4.3 **Other miscellaneous functions – anymissing(), nameof(), error() and gettime()**
When x is REAL or LOGICAL, anymissing(x) has value True if and only if at least one element of x is MISSING. If x is CHARACTER, anymissing(x) has value True if and only if at least one element of x is "". Unlike the other functions in this section, anymissing() accepts only one argument. The argument can, however, be a structure in which case anymissing() returns a structure with LOGICAL components.

```
Cmd> a <- matrix(vector(1,3,4,2,?,0,6,7),2) # note MISSING value
Cmd> anymissing(a)
(1) T
Cmd> anymissing(structure(a, b:structure(b1:"",b2:run(5))))
component: a
(1) T
component: b       structure component of a structure
  component: b1
(1) T
  component: b2
(1) F
```

Function nameof() returns the names of its arguments as a CHARACTER vector.

```
Cmd> nameof(x,cos,boxcox,run(5),"hello",F)
(1) "x"
(2) "cos"
(3) "boxcox"
(4) "VECTOR"
(5) "STRING"
(6) "LOGICAL"
```

Command error() works almost identically to print() except that it signals to MacAnova an error has occurred, terminating the macro. An example of its use is the following fragment from a macro

```
@arg1$$ <- $1
if(!isreal(@arg1$$)){error("ERROR: argument 1 must be REAL")}
```

Since you use error() only to print error messages, when it prints a single CHARACTER scalar or quoted string which doesn't start with "ERROR: ", it inserts "ERROR: " before

the message.

```
Cmd> error("Test of error()") # message doesn't start with ERROR:
ERROR: Test of error()     Printed message starts with ERROR:
```

Function gettime() allows you to time commands. It has several usages, controlled by keywords interval, quiet, and keep. These are best illustrated by example.

```
Cmd> gettime() # prints time since start of run
Time since start is 377.65 seconds

Cmd> gettime(interval:T) # prints time since last use of gettime()
Elapsed time is 1.1 seconds

Cmd> gettime(quiet:T) #or gettime(interval:T,quiet:T)

Cmd> gettime(interval:T)
Elapsed time is 2.2 seconds

Cmd> time <- gettime(keep:T); time# return cumulative time as value
(1)       392.98

Cmd> d <- gettime(interval:T,keep:T,quiet:F)
Elapsed time is 2.67 seconds

Cmd> d
(1)        2.67
```

You can use gettime() to create a macro that will print the elapsed time of a macanova command or sequence of commands:

```
Cmd> timeit <- macro("gettime(quiet:T);{$0};gettime(interval:T)")

Cmd> timeit(x <- rnorm(10000);stuff<-describe(x))
Elapsed time is 0.16667 seconds
```

Note the use of $0 to replicate the entire argument of timeit. See Sec. 9.3.4.

In timeit, after whatever is done by $0, gettime(interval:T) prints the elapsed time since gettime(quiet:T) was executed before whatever is done by $0. This should be close to the time it took to execute $0.

On most computers, gettime() returns the actual time elapsed as might be measured with a stop watch. On a few computers, the time is the amount of central processor time used. This will generally be less, often much less than the actual elapsed time.

9.4.4 **Keywords in macros – using $K, $k** Macro symbols $K and $k (Sec. 9.3.4) allow you to isolate the keywords in a macro's arguments. In many cases that is all that is needed, since all keywords will be passed on to a single MacAnova function. This is the case with pre-defined macro colplot whose text is similar to the following.

```
if($N < 1){error("$S expects at least 1 argument")}
chplot(1,$1,lines:T,$K,xlab:"Row Number")
```

Here $K is included in the argument list to chplot(). Following it is a default value for keyword xlab which is operative only if the user of the macro does not supply labels for the X-axis. If no keywords are supplied so that $K expands to nothing, chplot() will have an "missing" argument. Since this is a common usage with

plotting commands, they do not consider this to be an error.

Most other commands *do* consider a missing argument to be an error. In such cases, the macro should use different "calls" to the function depending on whether or not `$k` is 0. We illustrate this by writing a simple macro to sort a matrix across each row. If `down:T` is an argument, each row will be in descending order. If not, we want each row to be in ascending order. Note the two uses of the transpose operator "`'`", first to change rows into columns that `sort()` can operate on, and then to change them back to rows after the sort.

```
Cmd> sortrows <- macro("sort(($1)',$K)'") # 1st try

Cmd> data <- matrix(vector(28.4,21.6,23.1,22.1,\
    18.0,20.4,24.5,24.8),2) # small 2 by 4 matrix

Cmd> data
(1,1)         28.4         23.1           18         24.5
(2,1)         21.6         22.1         20.4         24.8

Cmd> sortrows(data,down:T)# works as we hope
(1,1)         28.4         24.5         23.1           18
(2,1)         24.8         22.1         21.6         20.4

Cmd> sortrows(data) # no keywords; hope to sort rows "up"
ERROR: argument 2 to sort is missing
```

It didn't work without the keyword. Here is an improved version.

```
Cmd> sortrows<-macro("if($k!=0){sort(($1)',$K)'}else{sort(($1)')'}")

Cmd> sortrows(data) # new version works without keyword
(1,1)           18         23.1         24.5         28.4
(2,1)         20.4         21.6         22.1         24.8
```

If `sortrows` is to be used a lot, we would want to add additional lines to check that the arguments are appropriate (see Sec. 9.4.2, 9.4.5). Note that `sortrows` makes use of the fact that the value of `if(...){...}else{...}` is the value of whichever compound command is actually executed. See. Sec. 9.2.2.

In other situations, you may want to use keywords to control what happens in the macro itself. One way to do this is to create a structure consisting of all the keyword values and then use `match()` (Sec. 9.4.1) to check whether a keyword was used. Suppose, for example, that a macro is supposed to recognize optional keywords `left` and `right` with default `LOGICAL` values False. The following macro fragment indicates how this might be done.

```
@left$$ <- @right$$ <- F
if ($k > 0){
    @keys$$ <- structure($K)
    @j$$ <- match("left",compnames(@keys$$),0)
    if(@j$$>0){@left$$ <- @keys$$[@j$$]}
    @j$$ <- match("right",compnames(@keys$$),0)
    if(@j$$>0){@right$$ <- @keys$$[@j$$]}
    ... check that values are LOGICAL scalars using isscalar()...
}
```

Because argument 3 of `match()` is 0, `match()` returns 0 if it cannot match a `"left"` or

"right" among the component names of @keys$$. If a match is made, the value of match() gives the component number so that the value can be extracted from @keys$$.

9.4.5 **Checking and evaluating keyword phrase arguments – keyvalue()** When you write a macro for use by others, it is essential that the macro check its arguments fairly thoroughly, printing informative error messages when necessary. Although this can be done using only functions like isvector() and isreal() described in Sec. 9.4.2, together with clever use of match() as illustrated in Sec. 9.4.4, it is often easier to use keyvalue() to evaluate and check keyword phrase arguments and argvalue() (see Sec. 9.4.6) to evaluate and check ordinary arguments, respectively.

keyvalue() provides a direct way to "parse" keyword phrases, simultaneously determining if a keyword has been used in the argument list, checking properties such as type, shape and sign of its value, and, if there is no error, returning value of the keyword. The general usage is

```
value <- keyvalue(name1:value1,name2:value2, ...,KeyName,Properties)
```

KeyName is a quoted string or CHARACTER scalar which specifies the keyword looked for and Properties is a CHARACTER scalar or vector which specifies properties that the value of the keyword must have if it is present. A typical value for Properties is vector("integer","vector","positive"), which you might use when checking a keyword phrase whose value must be a vector of positive integers. You actually need only the first three letters of each property (4 letters for "nonnegative" and "nonmissing") so that vector("int","vec","pos") would mean the same thing. See Sec. 9.4.7 for details on allowable properties and which properties may be used together.

keyvalue() looks for KeyName among the keyword names name1, name2, ... When a match is found and the corresponding keyword value has all the required properties, the value is returned. When no match is found, keyvalue() returns NULL. When a matching name is found, but the value does not have all the required properties, it is an error which will terminate the macro in which keyvalue() is used.

```
Cmd> val <- keyvalue(a:10,b:20,"b","real");val# match w/correct type
(1)           20

Cmd> val <- keyvalue(a:10,b:20,"a","real");val# match w/correct type
(1)           10

Cmd> val <- keyvalue(a:10,b:20,"c","real");print(val)# no match
val:
(NULL)

Cmd> val <- keyvalue(a:10,b:20,"a","logic") # match with wrong type
ERROR: value of keyword 'a' is not LOGICAL

Cmd> val <- keyvalue(a:matrix(run(4),2),"a",vector("real","vector"))
ERROR: value of keyword 'a' is not a REAL vector
```

An alternate usage is

```
value <- keyvalue(structure(name1:value1,...), KeyName, Properties)
```

for which the component names of the first argument are scanned for a match to KeyName.

```
Cmd> val <- keyvalue(structure(a:10,b:20),"b","real"); val
(1)           20
```

There must be at least 1 argument before KeyName. If the first argument is empty (for example keyvalue(,"a","real")), keyvalue() returns NULL.

You can use "wild card" characters "*" and "?" in KeyName to allow for some variation in keyword spelling. In seeking a keyword name matching KeyName, "*" matches 0 or more consecutive characters, without regard to what they are, and "?" matches any single characters. Thus when KeyName is "pow*", it will match keywords power, pow and powers among many possibilities. When KeyName is "m??imum", it would match keywords minimum or maximum. Judicious use of these wild card characters allows for more user friendly macros.

Here is how the macro fragment near the end of Sec. 9.4.4 might have been written using keyvalue().

```
@left$$ <- keyvalue($K,"left",vector("logic","scalar"))
@right$$ <- keyvalue($K,"right",vector("logic","scalar"))
if (isnull(@left$$){@left$$ <- F}#set default
if (isnull(@right$$){@right$$ <- F}#set default
```

This automatically checks that the values of left and right, if present, are LOGICAL and supplies default values False when they are not. This works even when there are no keywords in the argument list, because in that case keyvalue($K,"left", "logic") expands as keyvalue(,"left","logic") which has value NULL.

Yet another form for the fragment would be

```
@left$$ <- @right$$ <- NULL
if ($k > 0){
    @keys$$ <- structure($K);
    @left$$ <- keyvalue(@keys$$,"left",vector("logic","scalar"))
    @right$$ <- keyvalue(@keys$$,"right",vector("logic","scalar"))
}
if (isnull(@left$$){@left$$ <- F}
if (isnull(@right$$){@right$$ <- F}
```

This has the advantage that the keyword phrases are evaluated only once when executing structure(), while the previous fragment evaluated them both times keyvalue() was executed.

9.4.6 **Checking and evaluating non-keyword arguments – argvalue()** You can use argvalue() in a macro to evaluate a non-keyword phrase argument and at the same time check that it has specified properties. The general usage is

```
value <- argvalue(arg, ArgName, Properties)
```

where arg is an arbitrary variable of expression, ArgName is a quoted string or CHARACTER scalar, and Properties is a CHARACTER scalar or vector of the same sort used in keyvalue() (See Sec. 9.4.5 and 9.4.7). Argument arg is checked to see if it has

all the properties specified by Properties. If it does, then the value of arg is assigned to value. If it does not, argvalue() reports an error which terminates the macro and prints an informative error message incorporating ArgName.

```
value <- argvalue(arg, ArgName) # no Properties
```

is equivalent to value <- arg except that ArgName is used in the error message when the assignment cannot be carried out, as when arg is not defined.

As a typical example of how argvalue() might be used, here is the text of macro gamma:

```
if ($v != 1 || $k > 0){error("usage is gamma(x)")}
@x$$ <- argvalue($1,"$1",vector("positive","array"))
exp(lgamma(@x$$))
```

This uses lgamma() to compute the gamma function of a REAL vector, matrix or array of positive elements. The first line checks that there is exactly one non-keyword argument and no keyword arguments (see Sec. 9.3.4) . The second line copies the single argument, but will print an error message when it is not a REAL array all of whose elements are positive (recall that scalars, vectors and matrices are all particular cases of arrays). Here are examples of how you might use gamma:

```
Cmd> gamma(run(5))
(1)           1           1           2           6          24

Cmd> gamma(run(0,2)) # illegal argument
ERROR: run(0,2) is not an array of positive REALs

Cmd> gamma(vector(3.5,8,?,2))
ERROR: vector(3.5,8,?,2) has MISSING elements
```

The error message in the last example was printed because property "positive" implies property "nonmissing". See Sec. 9.4.7.

9.4.7 **Properties checked by keyvalue() and argvalue()** Each permissible element of the CHARACTER vector Properties used as an argument for keyvalue() and argvalue(), may be classified as to whether it describes the *type*, *shape*, *value* or *sign* of a variable. Here is a table of all legal properties classified as to what kind they are:

Kind of property	Legal property names
Type	"real", "logic", "character", "macro", "graph", "notnull"
Shape	"scalar", "vector", "matrix", "array", "structure"
Value	"integer", "nonmissing"
Sign	"positive", "nonnegative"

There are some sensible restrictions on what properties can be used together:

- No more than one property of each kind can be specified.
- Properties "positive", "nonnegative" and "integer" imply properties "real", and "nonmissing" and can't be used with any Type property except "real".
- Property "nonmissing" is can't be used with any Type property except "real" and "logical".

- Property "structure" can't be used with any Sign or Value property.
- Properties "macro", "graph" and "notnull" cannot be used with any other property.

As mentioned previously, you can abbreviate any property to as few as its first three letters (four for "nonmissing" and "nonnegative").

9.5 **Indirect evaluation of commands** MacAnova has two ways of indirectly evaluating or executing MacAnova commandsbesides using a macro – function evaluate() and the syntactical construct <<...>>.

9.5.1 **evaluate()** When Cmds is a quoted string or CHARACTER scalar containing one or more MacAnova commands separated by semicolons, evaluate(Cmds) executes the commands and returns as its value the value of the last command executed.

```
Cmd> evaluate("print(\"Hello!\");sqrt(2*PI)")
Hello!
(1)       2.5066
```

evaluate(Cmds) behaves very much as if Cmds were an out-of-line macro (Sec. 9.3.5) and as such adds little additional functionality. evaluate() can be useful in a loop, creating several variables with different names:

```
Cmd> x <- run(5)

Cmd> for(i,run(3)){evaluate(paste("x",i," <- x^",i,sep:""));;}

Cmd> hconcat(x1,x2,x3)
(1,1)            1            1            1
(2,1)            2            4            8
(3,1)            3            9           27
(4,1)            4           16           64
(5,1)            5           25          125
```

Each time through the loop, evaluate() assigns a value to a variable. For example, when i = 2, the command executed by evaluate() is x2 <- x^2.

Evaluate can be used recursively.

```
Cmd> evaluate("evaluate(\"sqrt(2)\")/evaluate(\"sqrt(PI)\")")
(1)      0.79788
```

The combined depth of recursive uses of evaluate() and out-of-line macros cannot exceed 50.

9.5.2 **Indirect references using <<...>>** An alternative to evaluate(Cmd) is <<Cmd>>, where Cmd is again a CHARACTER scalar containing one or more MacAnova commands. In most contexts this is entirely equivalent to evaluate(Cmd).

```
Cmd> <<"print(\"Hello!\");sqrt(2*PI)">>
Hello!
(1)       2.5066

Cmd> <<"3.14159">>^.5 # or <<"3.14159^.5">>
(1)       1.7725
```

```
Cmd> print(<<"T">>,<<"-123.45">>,<<"NULL">>,<<"\"MacAnova\"">>)
LOGICAL:
(1) T
NUMBER:
(1)      -123.45
NULL:
(NULL)
STRING:
(1) "MacAnova"
```

You would get the identical output from `print(T,-123.45,NULL,"MacAnova")` or `print(evaluate("T"),evaluate("-123.45"),evaluate("NULL"), evaluate("\"MacAnova\""))`.

`<<Cmd>>` behaves slightly differently when `Cmd` is the name of a variable, a macro, a function or a structure component. In that case it is interpreted as an indirect reference to the object named. For example, `<<"cos">>(PI/6)` is entirely equivalent to `cos(PI/6)` and `<<"a">> + <<"b">>` is equivalent to `a + b`.

```
Cmd> vector(<<"cos">>(PI/6), cos(<<"PI">>/6))
(1)      0.86603      0.86603

Cmd> x <- vector(9.53,5.59,9.27,7.19,10.98)

Cmd> <<"print">>(<<"boxcox">>(x,.5))#indirect refs to print & boxcox
VECTOR:
(1)       12.013        7.853       11.769       9.6783       13.317

Cmd> temperatures$<<"Sunday">> # same as temperatures$Sunday
(1)           61           73           85           83           81
```

(See Sec. 2.8.16 for the information about structure `temperatures`.)

On a Macintosh you can use the characters « and » (Option-\ and Option-|) in place of << and >>.

9.6 **Analysis of macro regs** We illustrate the use of `<<...>>` as well as other macro-related features by examining in detail the pre-defined macro `regs`. If `x` is a matrix and `y` is a vector or matrix with the same number of rows as `x`, `regs(x,y)` computes the regression of `y` as response variable on the columns of `x` as independent variables.

Here is `regs`, with added line numbers for easier reference:

```
    Cmd> print(paste(regs))
 1 #regs(x,y),matrix or vector y, matrix x
 2 @Xvars$$<-$1
 3 @Y<-$2
 4 if(!ismatrix(@Xvars$$)||!ismatrix(@Y)){
 5   error("usage: $S(x,y), matrix x, vector or matrix y")
 6 }
 7 @p$$<-length(@Xvars$$)/dim(@Xvars$$)[1]
 8 @X1<-@Xvars$$[,1]
 9 STRMODEL<-"@Y=@X1"
10 if(@p$$>1){
11   for(@i$$,run(2,@p$$)){
12     <<paste("@X",@i$$,sep:"")>><-@Xvars$$[,@i$$]
13     STRMODEL<-paste(STRMODEL,"+@X",@i$$,sep:"")
```

```
14   }
15 }
16 if(length(@Y)==dim(@Y)[1]){
17   regress()
18 }else{
19   manova()
20   print("NOTE: use secoefs() to get coefficients and standard
   errors.")
21 }
```

It is not necessary to include a comment summarizing a macro's usage, as is done in line 1, but is a good idea. Since lines starting with "#" are recognized by `macrousage()` the user has easy access to them. See Sec. 2.9.3. If you used `$S` instead of the macro name (here `#$S(x,y) . . .`), `macrousage()` will substitute the macro name when printing it.

Lines 2 and 3 copy `x` and `y` to temporary variables `@Xvars$$` and `@Y`. Name `@Xvars$$` will expand to something like `$Xvars50`. See Sec. 9.3.3.

Lines 4–6 check the arguments and terminate the macro with an informative message if they are not both matrices. See Sec. 9.4.2. A more thorough check would use `ismatrix(@Xvars$$,real:T)` and `ismatrix(@Y,real:T)`, and check that `nrows(x)` = `nrows(y)`.

Lines 2 through 6 might be replaced by

```
@Xvars$$ <- argvalue($1,"$1",vector("real","matrix"))
@Y$$ <- argvalue($2,"$2",vector("real","matrix"))
```

although no message giving the correct usage would be printed. See Sec. 9.4.6.

Line 7 obtains the number of columns of `x` and should probably have been written `@p$$ <- ncols(@Xvars$$)`.

Line 8 creates variable `@X1` from column 1 of `x`.

Lines 9 - 15 construct `STRMODEL` so as to have the form `"@Y=@X1+@X2+@X3+..."`. In addition, they create temporary variables `@Y` from `y`, and `@X1`, `@X2`, ... from the columns of `x`.

If the test in line 10 indicates that `x` has more than one column, line 11 starts looping over columns 2 through `@p$$` of `x`. As `@i$$` loops through 2, 3, ... , `paste(\"@X\",@i$$,sep:\"\")` (line 12) creates `CHARACTER` scalars with values `"@X2"`, `"@X3"`, ... Thus, for example, when `@i$$` is 2,

```
<<paste("@X",@i$$,sep:"")>> <- @Xvars$$[,@i$$]
```

is equivalent to

```
<<"@X2">> <- @Xvars$$[,2]
```

which in turn is equivalent to

```
@X2 <- @Xvars$$[,2]
```

creating vector `@X2`. Similarly in line 12, the strings `"+@X2"`, `"+@X3"`, ... are appended to `STRMODEL` one by one.

In lines 16 through 21 are the actual linear model computations . These use `regress()` when `y` is a vector and `manova()` when `y` has more than 1 column. In the latter case, the user is reminded that `secoefs()` must be used to see the coefficients and their standard errors.

9.7 **User functions** A user function is a program or program fragment compiled separately from MacAnova that is written in such a way that it can be executed from within MacAnova using functions `loadUser()` and `User()`. User functions can "call back" to MacAnova to execute MacAnova commands, and on most systems they can be written in such a way as to have automatic checking of arguments. User functions may be written in the C programming language and in some cases in Fortran.

Information on programming user functions and details on their use on different computer systems are beyond the scope of this manual, but will be provided in a yet to be completed separate document. Fairly complete information is currently available in file Userfun.hlp distributed with MacAnova. Here we limit discussion to a description of how `loadUser()` and `User()` are used. Briefly, `loadUser()` loads or "attaches" a file containing the user function in such a way that `User()` can execute the user function.

9.7.1 **loadUser()** In order to execute a user function, you must use `loadUser()` to inform MacAnova where to find it.

`loadUser(FileName)` loads ("attaches") file `fileName` and makes any user functions in it available to MacAnova. `FileName` should be a quoted string or `CHARACTER` scalar. Once the file is loaded, you execute user functions in it with `User()` (Sec. 9.7.2). As usual, in windowed versions (Macintosh, Windows, Motif), `FileName` can be "". If the file has been previously loaded, it is not reloaded, although it may be put at the start of a search list for the next use of User().

`loadUser(FileName,reload:T)` does the same, except that the file will be reloaded if it has been previously loaded into MacAnova.

`loadUser(FileName,clear:T)` does the same, except all previously loaded files will be forgotten.

The user function file must be of a special type specific to the MacAnova version. See topic `loadUser()` in file Userfun.hlp for information on file types (in MacAnova, type `help(file:"Userfun.hlp",loadUser)`).

9.7.2 **User()** The actual execution of a user function is controlled by `User()`.

`User(FuncName,arg1,arg2, ...)` executes a user function with name specified by quoted string or `CHARACTER` scalar `FuncName` and user function arguments `arg1`, `arg2`, ... Code for the user function must be in a file previously loaded by `loadUser()` (Sec. 9.7.1). You must have at least one user function argument and no more than 20 (13 in the Macintosh PPC version). What you should use for `FuncName` depends on the particular version of MacAnova. See topic `User()` in file Userfun.hlp for details. See Sec. 9.7.3 for details about user function arguments and the value returned by `User()`.

`User(FuncName,quiet:T,arg1, ...)` does the same except that any warning

messages are suppressed.

`User(FuncName,callback:T,arg1, ...)` specifies that the user function is known to "call back" to MacAnova, that is, to execute functions internal to MacAnova. See topic `callback_fun` in file Userfun.hlp.

`User(FuncName,symbols:T,arg1, ...)` specifies that all the arguments are to be passed as "symbols", an internal MacAnova data format which encapsulates the data, type, and dimensions of a variable. This should be used only with a user function specifically written to make use of MacAnova symbols.

`User(FuncName,resource:ResName, arg1, ...)` is needed on a Power Macintosh when the name of the user function differs from the name of the resource containing the function. `ResName` is a quoted string or `CHARACTER` scalar specifying the resource name.

`User(FuncName,pointers:T or F,arg1, ...)` changes the default way arguments are passed, either as "pointers" (`pointers:T`) or as "handles" (`pointers:F`). On all but Macintosh computers, the default is `pointers:T`. You are unlikely ever to use `pointers` since the default is usually appropriate.

You can use more than one of the preceding keywords phrases together as in `User("goo",resource:"foo",quiet:T,callback:T,x,result:0)`.

On most systems, a user function optionally may have an associated "arginfo" function that MacAnova can call to obtain information about the user function. When an arginfo function exists, user function arguments will be checked as to number and type and you shouldn't need to use keywords `callback`, `symbols` or `pointers`.

9.7.3 **User function arguments and value returned** All arguments to `User()` are user function arguments with the exception of the file name and `quiet`, `callback`, `symbol`, `pointers` or `resource` keyword phrases . They provide input to the user function and provide a place for it to put its results. For conciseness, in this section "user function argument" is shortened to "argument."

Except when `symbols:T` is an argument to `User()`, all arguments must be either `REAL`, `LOGICAL`, `CHARACTER` or `LONG` variables. `REAL` arguments are passed as double precision data (C `double`, Fortran `REAL *8`), as are `LOGICAL` arguments (True = 1.0, False = 0.0). `LONG` variables may be created by function `asLong()` (Sec. 9.7.4) and have only a transitory existence as the arguments to `User()`. Their values are signed integers between $-2^{31}+1$ and $2^{31}-1$. `CHARACTER` arguments are passed as character vectors (C `char []`). If a `CHARACTER` argument argument is a vector, matrix or array, the individual elements follow one another, each terminated by a null character (`'\0'`). Since this is based on the standard form of strings in C, it may not be possible to use `CHARACTER` arguments with a user function compiled in Fortran.

If an argument is a matrix or array, the values are ordered such that the first subscript changes fastest.

Two user functions, `add1` and `innerprod`, are used as examples. `add1` has three real scalar arguments, say *a*, *b*, and *c* and performs the computation $c = a + b$. `innerprod` has four arguments, say, *x*, *y*, *n* and *s*, where *n* is a positive integer (type `LONG`), *x* and *y* are `REAL` vectors of length *n* and *s* is a `REAL` scalar. It performs the computation $s = \sum_{i=1}^{n} x_i y_i$.

An argument that is not a keyword phrase is passed directly to the user function without being copied. If it is a named MacAnova variable and the user function modifies that argument, the value of the variable itself is changed.

```
Cmd> c <- 0; User("add1",3,sqrt(25),c)

Cmd> c # c has a new value
(1)             8
```

If the argument is a literal number or expression as is the case with the first two arguments in the example, the function can safely change the argument without danger to any variable as long it does not go beyond the size of the variable, that is, with a `REAL` or `LOGICAL` argument `arg`, no more than `length(arg)` elements are modified.

Keyword phrase arguments are used for two purposes – to protect the argument passed from modification by the user function and to specify what values `User()` returns.

Only a *copy* of an argument that is a keyword phrase value is passed to the user function. This means that the user function can modify such an argument with no danger of changing any MacAnova variable, again as long as it respects the size of the variable.

The values of keyword arguments, possibly modified by the user function, are returned as the value of `User()`.

```
Cmd> c <- 0; User("add1",3,5,result:c)
(1)             8   Value returned by User()

Cmd> c # c has a original value
(1)             0
```

When more than one argument is a keyword phrase, `User()` returns a structure, with one component for each keyword phrase:

```
Cmd> c <- 0; User("add1",a:3,b:5,result:c)
component: a
(1)             3
component: b
(1)             5
component: result
(1)             8

Cmd> User("innerprod",x:run(5),y:vector(3,1,10,2,4),n:asLong(5),s:0)
component: x
(1)             1             2             3             4             5
```

```
component: y
(1)            3            1           10            2            4
component: n
(1)            5
component: s
(1)           63
```

Keyword `protect` is special in that its value will *not* be returned although it will be copied before being passed to the user function. For example, a function that computes a median from a vector `x` might first sort `x` and then find the middle value of the sorted vector. If you want to preserve the original ordering of `x`, you should pass it to the function using keyword phrase `protect:x`.

As mentioned above, when a user function modifies an argument it is essential that the argument be long enough to hold all the changes. For example, suppose `add4` is a user function similar to `add1` except its arguments are expected to be vectors of length 4, with the third argument set to the sum of the first two arguments.

```
Cmd> User("add4",run(4),vector(3,1,0,2),result:rep(0,4))
(1)            4            3            3            6
```

But

```
Cmd> User("add4",run(4),vector(3,1,0,2),result:0)
```

will probably result in MacAnova crashing, because room for only one number was provided as value for `result` and `add4` will try to change four numbers.

Often, as here, the value of an argument that a user function modifies has no purpose other than to provide space for the user function to put its answer.

When keyword phrase `symbols:T` is an argument to `User()`, *all* user function arguments are passed as *symbols* (see Sec. 9.7.2). You cannot have some arguments be symbols and some just data

9.7.4 **Passing integer arguments – aslong()** Many C or Fortran functions expect integer arguments (type `int` or `long` in C and `INTEGER` in Fortran). To accommodate this need, at least partially, user function arguments to `User()` can be of the form `asLong(n)`, where `n` is a variable of type `REAL` all of whose elements are integers with values between $-2^{31}+1$ and $2^{31}-1$. A transitory variable of MacAnova type `LONG` is created and passed to the function. If the user function is written in C, the corresponding argument must be declared as a 32 bit integral type. This will usually be type `int` or `long`, depending on the computer or compiler. A user function written in Fortran would normally declare the argument to be `INTEGER*4`.

If you assign the value of `asLong()` to a variable, it gets automatically "coerced" back to a `REAL` variable.

```
Cmd> a <- asLong(run(-2,2))
Cmd> list(a)
a                REAL    5
```

The same thing happens when a user function argument is a keyword phrase. The

value is coerced to REAL before returning.

```
Cmd> s <- 0; n <- User("innerprod",x,y,n:asLong(5),s); list(n)
n               REAL   1
```

You can't use LONG data created by asLong() in arithmetic or, with the exception of User(), print() and write(), as the argument to a function.

```
Cmd> 3 + asLong(5)
ERROR: arithmetic with non-numeric and non-logical operand
REAL + LONG near 3 + asLong(5)

Cmd> sum(asLong(run(5)))
ERROR: argument to sum must not have type LONG

Cmd> print(asLong(run(-2,2)))#prints as if REAL
VECTOR:
(1)           -2           -1            0            1            2
```